Design and Analysis of Data Structures for Dynamic Trees

Renato F. Werneck

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

June, 2006

Abstract

The dynamic trees problem is that of maintaining a forest that changes over time through edge insertions and deletions. We can associate data with vertices or edges and manipulate this data, individually or in bulk, with operations that deal with whole paths or trees. Efficient solutions to this problem have numerous applications, particularly in algorithms for network flows and dynamic graphs in general. Several data structures capable of logarithmic-time dynamic tree operations have been proposed. The first was Sleator and Tarjan's ST-tree, which represents a partition of the tree into paths. Although reasonably fast in practice, adapting ST-trees to different applications is nontrivial. Frederickson's topology trees, Alstrup et al.'s top trees, and Acar et al.'s RC-trees are based on tree contractions: they progressively combine vertices or edges to obtain a hierarchical representation of the tree. This approach is more flexible in theory, but all known implementations assume the trees have bounded degree; arbitrary trees are supported only after ternarization. This thesis shows how these two approaches can be combined (with very little overhead) to produce a data structure that is at least as generic as any other, very easy to adapt, and as practical as ST-trees. It can be seen as a self-adjusting implementation of top trees and provides a logarithmic bound per operation in the amortized sense. We also discuss a pure contraction-based implementation of top trees, which is more involved but guarantees a logarithmic bound in the worst case. Finally, an experimental evaluation of these two data structures, including a comparison with previous methods, is presented.

Acknowledgements

I am deeply indebted to my advisor, Bob Tarjan, for his guidance and patience. I thought highly of him before I came to Princeton, but now I realize it was not nearly enough. Working with him was a privilege and, above all, a pleasure.

I thank my readers, Adam Buchsbaum and Bernard Chazelle, for their numerous comments on this dissertation. I also thank Robert Sedgewick and Nicholas Pippenger for taking their time to participate in the thesis committee, and for their questions and suggestions during the presentation of the thesis proposal.

I had important discussions about dynamic trees with Umut Acar, Guy Blelloch, Jorge Vittes, and Loukas Georgiadis. I thank Phil Klein for his many helpful comments on the self-adjusting data structure presented in Chapter 4. Mikkel Thorup and Jakob Holm are coauthors (with Bob Tarjan and I) of the worst-case data structure presented in Chapter 3.

Diego Nehab has provided me with the computational resources for conducting the experiments described in Chapter 5 (i.e., he let me borrow his computer). Kevin Wayne generously taught me how to use the binding machine, all in exchange for a mere acknowledgement. I am eternally indebted to him.

I thank everybody at Microsoft Research Silicon Valley for giving me the peace of mind to write this dissertation. There is nothing like having a job.

I would also like to thank my coauthors in projects I worked on while at Princeton but that are not part of this dissertation. In particular, I thank Ricardo Fukasawa, Loukas Georgiadis, Andrew Goldberg, Haim Kaplan, Jens Lysgaard, Marcus Poggi de Aragão, Mauricio Resende, and Eduardo Uchoa. As great as dynamic trees are, it is always nice to work on other topics.

I thank the administrative staff at the Computer Science department, in particular Melissa Lawson and Mitra Kelly, for shielding me from the bureaucracy of real life.

During more than a decade as a Computer Science student, I was fortunate to have great mentors and advisors: João Carlos Setubal at Unicamp, Marcus Poggi de Aragão at PUC-Rio, Mauricio Resende at AT&T Labs Research, Andrew Goldberg at Microsoft Research Silicon Valley, and, of course, Bob Tarjan at Princeton. I cannot thank them enough for their guidance.

On a personal note, I thank my friends for making my years at Princeton unforgettable. This includes Adrian, Diego, Diogo, Loukas, Thomas, Tony, and especially anyone who is considering not talking to me anymore just because I did not mention you by name.

Most importantly, I dedicate this dissertation to my parents, Dorothea and Rogério. Without them I would be nothing—literally.

My work was funded by Princeton University and the Aladdin Project (National Science Foundation grant no. CCR-0122581). Additional summer funding (through internships) was provided by AT&T (in 2002 and 2003) and Microsoft (in 2004 and 2005).

Contents

Abstract **iii**

Acknowledgements **iv**

Contents **vi**

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

2 Existing Data Structures **5**

2.1 Path Decomposition 6

2.1.1 Representation 6

2.1.2 Updating the Tree 8

2.1.3 Dealing with Values 10

2.1.4 Undirected Trees 11

2.1.5 Aggregating Information over Trees 12

2.1.6 Other Extensions 15

2.2 Tree Contraction 16

2.2.1 The Parallel Setting 16

2.2.2 Topology Trees 18

2.2.3 RC-Trees 21
2.2.4 Top Trees 23
2.3 Euler Tours 34

3 Contraction-Based Top Trees 38

3.1 Number of Levels 39
3.2 Updating the Contraction 44
3.2.1 Updates: Basic Notions 47
3.2.2 Proof Outline 51
3.2.3 Replicated Moves 53
3.2.4 Stable Subtours 55
3.2.5 Unstable Subtours 61
3.2.6 Running Time 67
3.3 Implementation 71
3.3.1 Representation 71
3.3.2 Identifying Valid Moves 72
3.3.3 Updating the Tree 72
3.3.4 Other Details 75
3.3.5 Implementing Expose 76
3.4 Alternative Design Choices 80
3.4.1 No Circular Order 80
3.4.2 Back Rakes 82
3.4.3 Alternating Rounds 83
3.4.4 Randomization 83

4 Self-Adjusting Top Trees 85

4.1 Representation 85
4.1.1 Order within Binary Trees 90
4.1.2 Handles 90

4.2 Updates 91
4.2.1 Soft Expose 91
4.2.2 Hard Expose 97
4.2.3 Cuts 98
4.2.4 Links 99
4.2.5 Implementation Issues 100
4.3 Analysis 101
4.4 Alternative Representations 105
4.4.1 Possible Simplifications 105
4.4.2 Unit Trees 106
4.5 Path Decomposition and Tree Contraction 107
4.5.1 Contraction to Decomposition 108
4.5.2 Decomposition to Contraction 109
4.6 Final Remarks 110

5 Experimental Analysis 111
5.1 Experimental Setup 112
5.2 Data Structures 112
5.2.1 ST-trees 113
5.2.2 ET-trees 115
5.2.3 Top Trees 115
5.3 Maximum Flows 117
5.3.1 Basics 118
5.3.2 The Shortest Augmenting Path Algorithm 119
5.3.3 Experimental Results 124
5.4 Online Minimum Spanning Forests 128
5.4.1 The Algorithm 128
5.4.2 Experimental Setup 130
5.4.3 Random Graphs 131

5.4.4 Circular Meshes . . . 135

5.4.5 High-Degree Vertices . . . 138

5.4.6 Memory Usage and Cache Effects . . . 141

5.5 Single-Source Shortest Paths . . . 143

5.5.1 Algorithm . . . 144

5.5.2 Experiments . . . 146

5.6 Random Operations . . . 148

5.7 Previous Work . . . 151

5.8 Final Remarks . . . 154

6 Final Remarks **156**

References **160**

List of Figures

2.1 Example of an ST-tree 7

2.2 Ternarization 13

2.3 Topology tree 18

2.4 A free tree 24

2.5 Top trees: Basic operations 24

2.6 A contraction and the corresponding top tree 26

2.7 A top tree without dummy nodes 26

2.8 An Euler tour 36

2.9 A binary tree representing an Euler tour 36

3.1 Bad configurations 41

3.2 Good configurations 42

3.3 A maximal set of moves that eliminates only 1/6 of the vertices 45

3.4 A tree that requires two rounds to have half of its vertices eliminated 46

3.5 Updating a contraction: An example 48

3.6 An Euler tour as used by top trees 49

3.7 Proper subtours 50

3.8 A new move that involves no extreme arc 55

3.9 Stable Subtours 56

3.10 Unstable Subtours 62

3.11 Implementing *expose* with a temporary tree 79

3.12 Why pairing up edges arbitrarily does not work 81

4.1 A unit tree 86

4.2 An augmented top tree 87

4.3 Complete example 88

4.4 Augmented top tree corresponding to Figure 4.3 89

4.5 Actual top tree corresponding to Figure 4.3 89

4.6 Configuration after soft expose 92

4.7 Rotations in *compress* trees. 93

4.8 Splice: Free tree 94

4.9 Splice: Augmented top tree 95

4.10 Cutting 98

4.11 Linking 99

4.12 Splice: Phantom tree 102

5.1 Maximum flow on layered graphs: Relative times 125

5.2 Maximum flow on square meshes: Relative times 127

5.3 Online MSF: Random graphs with edges in random order 131

5.4 Online MSF: Operation counts on random graphs 133

5.5 Online MSF: Random graphs with edges in increasing order 134

5.6 Online MSF: Random graphs with edges in decreasing order 135

5.7 Online MSF: Symmetric circular meshes 136

5.8 Online MSF: Circular meshes with varying diameters 137

5.9 Extended star 139

5.10 Online MSF: Augmented stars with fixed number of vertices 140

5.11 Online MSF: Augmented stars with fixed spoke length 141

5.12 Online MSF: Cache effects 142

5.13 Single-source shortest paths: Running times 147

5.14 Random links and cuts 149

List of Tables

5.1 Maximum flows on layered graphs 125

5.2 Maximum flows on square meshes 127

5.3 Single-source shortest paths: Operation counts 148

Chapter 1

Introduction

Consider the following problem. We are given an n-vertex forest of rooted trees with costs on edges. Its structure can be modified by two basic operations: $link(v, w, c)$ adds an edge with cost c between a root v and a vertex w in a different component; $cut(v)$ removes the edge between v and its parent. At any time, we want to be able to find, for any vertex v, its parent $p(v)$ and the cost of the edge $(v, p(v))$. All these operations take constant time with an obvious implementation: with each vertex v, store a pointer to its parent and the cost of the edge between them.

Now suppose we also want to find the cheapest edge on the path from a vertex v to the root, or to add a constant c to the cost of every edge on this path. The obvious implementation can support these operations, but in time proportional to the length of the path, which could be $\Theta(n)$.

This specific problem appears in the context of network flow algorithms [51]. We are interested in its generalized version: a data structure to maintain a forest supporting in $O(\log n)$ time queries and updates related to vertices and edges individually, and to entire trees or paths. We call this the *dynamic trees problem.* Other typical operations include adding a certain value to all vertices in a tree, or asking for the sum of all edge weights on a path. Operations such as these are needed in several solutions to the maximum flow problem [6, 27, 29, 56] and related algorithms [51]. They are also used in algorithms that maintain properties of dynamic graphs, such as minimum spanning trees and connectivity [10, 24, 33, 35]. Applications for maintaining dynamic expression trees have also been reported [16, 25].

The first data structure to support every operation in the example application in $O(\log n)$ time was Sleator and Tarjan's ST-tree [51] (also known as the *link-cut tree*). This structure partitions the tree into vertex-disjoint paths and represents each one by a binary tree in which the original vertices appear in symmetric order. The binary trees are then glued together according to how the paths are connected. The root of each binary tree becomes a *middle child* of a node in another binary tree. For the algorithm to be efficient, this hierarchy must be balanced. But making each binary tree balanced is not enough—the total height of the hierarchy would be $O(\log^2 n)$. Sleator and Tarjan have shown that using globally biased search trees [13] one does achieve $O(\log n)$ worst-case time per operation. They later showed [52] how splaying greatly simplifies the data structure while still achieving the $O(\log n)$ bound (now amortized).

ST-trees can be adapted to solve other problems beyond our example, but this requires an understanding of their inner workings. In particular, Goldberg et al. [27] show how subtree-related operations (such as adding a value to all vertices in a tree) can be accomplished with an implicit ternarization of the original tree, which transforms high-degree vertices into chains of constant-degree ones. The data structure becomes more complicated, however.

A simpler and more elegant way to handle subtree-related operations stems from the observation that a tree can be represented by an Euler tour. Representing tours as standard balanced binary trees is the basis of ET-trees, proposed by Henzinger and King [33], and later simplified by Tarjan [56]. Unfortunately, these data structures cannot deal with path-related operations (such as the ones suggested in our example), so their applications are somewhat limited.

A third class of data structures is based on *tree contraction*. These structures use two operations proposed by Miller and Reif [43] in the context of parallel algorithms: *rake* (which removes leaves) and *compress* (which removes vertices of degree two). Each operation replaces the original elements (vertices and edges) by a *cluster* that aggregates information about them. The entire tree is represented by a hierarchy of clusters, which is itself a tree.

In Frederickson's *topology trees* [23, 24, 25], the contraction works in rounds, each with a maximal set of independent *rakes* and *compresses*. Since the tree shrinks by a constant factor in each round, there are $O(\log n)$ rounds. Furthermore, the contraction can be updated after a *link* or *cut* in $O(\log n)$ worst-case time. However, the data structure is somewhat involved, since it must maintain one tree for each level as well as the connections between these trees. In practical applications, this makes it considerably slower than ST-trees [25]. Recently, Acar et al. proposed *RC-trees* [3], a randomized variant that is conceptually simpler and runs in $O(\log n)$ expected time per operation. Both data structures view clusters as vertices, which, for technical reasons, means that the $O(\log n)$ bound only applies to trees of bounded degree. Arbitrary trees can be handled by ternarization, but this increases the tree size and adds an extra level of complexity.

An alternative is *top trees*, proposed by Alstrup et al. [7, 10]. By considering clusters to be edges (instead of vertices), they avoid the need for explicit ternarization. In addition, they provide an interface for handling data independently of the order in which *rakes* and *compresses* are performed, so one can adapt this data structure to different applications without modifying its inner workings (a similar interface was also defined for RC-trees). Not only does this simplify the implementation of existing algorithms for various applications, but it also makes it easier to devise new ones. In [10], however, the suggested implementation of top trees is as a layer on top of topology trees, hardly a practical solution.[1]

In a very broad sense, all these data structures have the same ultimate goal: to map an arbitrary tree into a balanced one. ET-trees do it in a very elegant, direct way, but they cannot deal with path-related operations. ST-trees represent individual paths as binary trees, which are then glued together to represent the whole tree. This approach is ideal for path-related operations, but handling subtree queries requires ternarization. Topology trees

[1]Holm and de Lichtenberg [34] did suggest a direct implementation of top trees, but they later found their run-time analysis to be flawed (personal communication). Even if the bound is correct, the implementation is far from trivial.

and RC-trees represent not the tree itself, but the steps necessary to contract it. This can be viewed as a multi-level decomposition of the original tree, which lends itself naturally to applications related to dynamic graphs. These two data structures, however, can only deal directly with trees of bounded degree. Top trees eliminate this constraint and have the most natural interface, but they achieve this by adding an extra layer to topology trees that merely hides the ternarization. Devising a data structure that is at the same time general, flexible, and practical has been an elusive goal.

This thesis achieves this goal. In Chapter 4, we show how the principles behind Sleator and Tarjan's ST-trees can be used to implement top trees. A partition of the original free tree into edge-disjoint paths can be directly mapped onto a series of *rakes* and *compresses*, which shows that partitions and contractions are essentially equivalent. The end result is a data structure that is almost as streamlined as the original ST-trees, but as flexible as top trees (with the extra ability to handle ordered edges around each vertex). Our data structure uses splaying and can handle dynamic tree operations in $O(\log n)$ amortized time.

We also present, in Chapter 3, a bottom-up, contraction-based algorithm for maintaining and updating top trees in $O(\log n)$ time per operation in the worst case. This is the first direct implementation of a worst-case top tree. (Recall that Alstrup et al.'s original proposal is to build an interface to topology trees.) Our algorithm consists of a very simple procedure to update the tree. Although it is not as practical as the self-adjusting version, it is much easier to describe and understand, and it does provide stronger running-time guarantees.

In Chapter 5, we present an experimental evaluation of our data structures. We compare the data structures to each other and to ET-trees and ST-trees (the first and still fastest of the dynamic trees data structures). The experiments uncover the strengths and weaknesses of each data structure, as well as of the top tree interface itself. Chapter 6 presents some concluding remarks. Before we deal with the new data structures, we provide (in Chapter 2) a more detailed description of previous methods.

Chapter 2

Existing Data Structures

This chapter presents an overview of existing data structures for maintaining dynamic trees. As already mentioned, all of them map arbitrary trees into balanced trees. They use three different approaches: path decomposition (ST-trees), tree contraction (topology trees, top trees, and RC-trees), and linearization (ET-trees). We discuss each approach in turn, in Sections 2.1, 2.2, and 2.3.

It should be noted, however, that these are not the only techniques used to map arbitrary trees into balanced trees. Tarjan [53], for example, used one such mapping in an algorithm that decides in $O(m\alpha(m,n))$ time whether a spanning tree of a graph with n vertices and m edges is minimum. Komlós [39] used a different mapping to show that a linear number of comparisons suffices to solve this verification problem, and his result was later used in actual linear-time algorithms by Dixon et al. [20] and King [37]. Chazelle [14] defines a more general canonical transformation that allows some data structures that work on lists to be applied to free trees. In particular, this transformation yields another simple $O(m\alpha(m,n))$-time algorithm for minimum spanning tree verification. All these mappings, however, apply to *static trees* only—edges cannot be added or removed.

We limit our discussion to dynamic trees. In particular, all data structures we discuss below can solve the *dynamic connectivity problem* for trees: they maintain a forest subject to edge insertions and deletions and support queries asking whether two vertices belong to the same tree or not. Pătraşcu and Demaine [45] showed that an $\Omega(\log n)$ lower bound in the cell-probe model applies to this problem. Mappings that deal with static trees are not subject to this lower bound, and indeed they can execute some operations faster than the data structures we describe in this chapter.

2.1 Path Decomposition

The first data structures to support dynamic tree operations in $O(\log n)$-time were Sleator and Tarjan's *ST-trees* [51, 52], also known as *link-cut trees* or simply *dynamic trees.* We will only use the term *ST-trees* in this dissertation to avoid any confusion, since the other two terms could easily apply to any data structure for this problem.

ST-trees are used primarily to represent rooted trees, with all edges directed towards the root. The version of ST-trees proposed in [52] associates a cost with each vertex in the forest. Costs are handled by the following operations:

- *findcost*(v): returns the cost of vertex v;
- *findmin*(v): returns the minimum-cost vertex on the path from v to the root of its tree (in case of ties, returns the one closest to the root);
- *addcost*(v, x): adds x to the cost of each vertex on the path from v to the root of its tree.

In addition, the data structure also provides a pair of operations to query the structure of the tree itself: *findroot*(v) returns the root of the tree containing v, and *parent*(v) returns the parent of v. Of course, ST-trees also support the usual structural operations: *link*(v, w) adds an edge between a root v and a vertex w in another tree, and *cut*(v) deletes the edge between v and its parent.

2.1.1 Representation

Each rooted tree in the forest is represented as follows. First, the tree is partitioned into vertex-disjoint paths. Edges within a path are called *solid* (and so is the path itself). The remaining edges, which link solid paths, are *dashed*. Each solid path is represented as a *solid subtree*, a binary search tree in which the original vertices appear in symmetric order: the bottommost vertex of the original path is represented as the leftmost vertex of the corresponding solid subtree.[1] Finally, the solid subtrees are "glued" together, creating a *shadow* (or *virtual*) tree. Let v be the topmost vertex of a solid path P, and let $p(v)$ be its parent (which belongs to some other path Q). The shadow tree represents this relationship as an edge between the root node r_P of the solid subtree representing P and the node representing $p(v)$; we say that r_P becomes a *middle child* of $p(v)$. There are no pointers from a node to its middle children; the pointers go from children to parent only. This is relevant because a node may have up to $\Theta(n)$ middle children. See Figure 2.1.

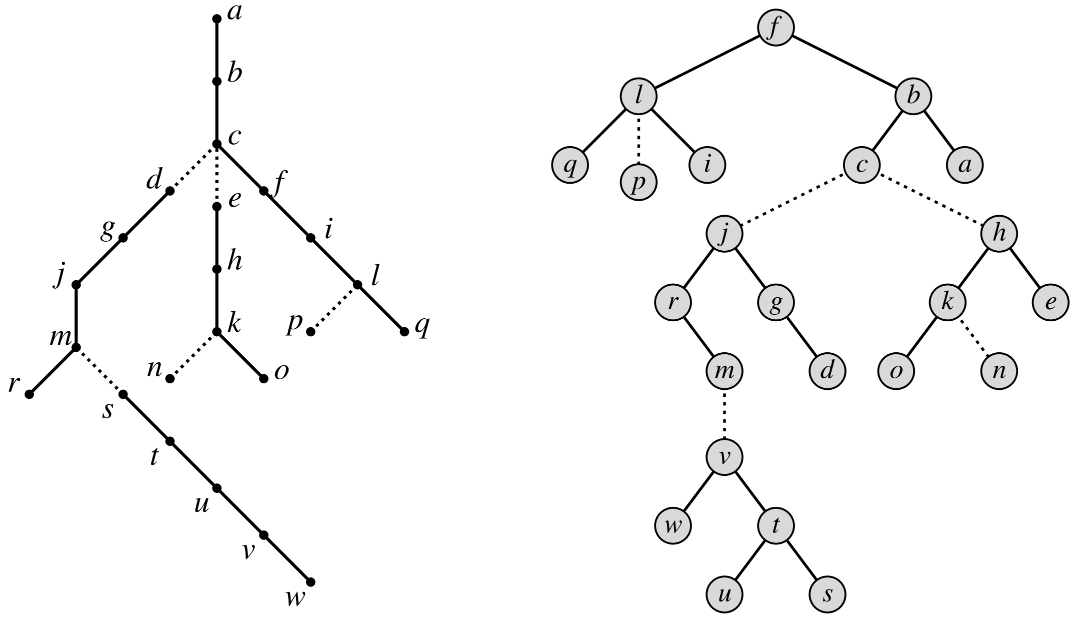


Figure 2.1: Example of an ST-tree (adapted from [52]). On the left, the original tree, rooted at a and already partitioned into solid paths; on the right, a shadow tree that represents it. In the shadow tree, middle children are connected to their parents by dotted lines. Values are not shown.

The implementation of the ST-tree operations described above is based on the *expose*

[1]This description follows the convention adopted in [52]. In [51], the order is reversed, which is equivalent but less natural.

operation, used internally only. Operation *expose*(v) ensures that the path from v to the root of its tree is solid and that there is no incoming solid edge into v. When these conditions hold, we say that v is the *exposed vertex* of the tree. Each tree in the forest has exactly one exposed vertex. In Figure 2.1, the exposed vertex is q. If one were to expose u instead, for example, edges (s, m) and (d, c) would be made solid, while (v, u), (r, m) and (f, c) would become dashed.

The conversion of edges from solid to dashed (and vice-versa) in the shadow tree is performed by the *splice* operation, used during *expose*. If v is the root of a solid subtree, *splice*(v) makes v the left solid child of its parent—it was formerly a middle child. If the parent had a left solid child already, it becomes a middle child. The left solid child is always the one replaced because the subpath it represents is always farther from the root than is the solid subpath represented by the right child. In fact, *splice*(v) can only be performed if the parent of v is itself on the left path of its solid subtree. In Figure 2.1, *splice*(p) would make p the left child of l in the shadow tree, and q would become l's middle child.

In addition to splices, *expose*(v) also needs to perform *rotations* within each solid subtree on the path from v to the root of the shadow tree. They are used to ensure that each solid subtree is properly balanced. These are standard binary tree rotations, and they only take solid children into account; the middle children of the nodes involved always remain attached to their original parents.

2.1.2 Updating the Tree

The running time of *expose* (and other structural operations) depends on what data structure is used to represent solid paths. The obvious candidates are balanced binary trees, such as red-black trees [31]. Unfortunately, the fact that each individual solid subtree is balanced does not guarantee that the whole structure will be. Each operation may take $\Theta(\log^2 n)$ amortized time, as shown in [51].

In [51], Sleator and Tarjan suggested the use of *biased search trees* instead. These are generalizations of balanced binary search trees (see [13, 22], for example) that allow *weights* to be assigned to the nodes. They ensure that the time to access a node with weight w is proportional to $\log(W/w)$, where W is the sum of all weights in the tree. Intuitively, nodes with higher weights are more likely to be closer to the root. These properties are relevant to ST-trees because the goal here is to make the entire structure balanced, not just each individual tree. This can be achieved by making the weight of each node equal to the number of descendants it has in the ST-tree structure, including those linked to it by dashed edges. Note that biased search trees are not as simple as weight-balanced search trees [44]: the latter is a standard binary search tree that uses the weight of a subtree (defined as the sum of the weights of its nodes) to keep the tree balanced, but every node has weight one. Biased search trees allow nodes to have arbitrary weights.

Sleator and Tarjan analyze the use of two different types of biased trees to implement ST-trees. *Globally biased search trees* can achieve $O(\log n)$-time per operation in the worst case, but they are extremely involved data structures. The authors suggest using *locally biased search trees* as an alternative. They are somewhat simpler and still guarantee an $O(\log n)$ bound per operation, but only in an amortized sense: starting from an empty tree, any sequence of m dynamic tree operations will take at most $O(m \log n)$ time, but some individual operations may take more than $\Theta(\log n)$ time. Unfortunately, locally biased search trees are still remarkably complicated data structures. For a description of both locally and globally biased search trees, see [13].

In [52], Sleator and Tarjan proposed a much simpler implementation of ST-trees, where solid paths are represented as *splay trees* (also introduced in [52]). These are self-adjusting binary search trees, and as such they maintain no balancing information whatsoever. Instead, whenever a node is accessed, it is *splayed*, i.e., it is brought to the root of its tree by a series of single and double rotations. The choice of which type of rotation to make in each step depends only on the local structure of the tree, but it ensures that each access to the splay tree will take logarithmic amortized time. In fact, an even stronger property holds: each access to an ST-tree, which involves splaying on a series of splay trees, will also take $O(\log n)$ amortized time.

2.1.3 Dealing with Values

An obvious way of dealing with values would be to store two pieces of information on each node of the shadow tree: $\mathit{cost}(x)$ would represent the cost of the node itself, and $\mathit{mincost}(x)$ the minimum cost of a descendant of x in the same solid subtree (including x itself).

To implement the *addcost* operation in $O(\log n)$ time, however, one cannot change the costs of all affected vertices explicitly. After all, a single path may have $\Omega(n)$ elements. To update values implicitly, values are represented in *difference form*. More precisely, each node x in the shadow tree stores two values:

- $\Delta\mathit{cost}(x)$: this is $\mathit{cost}(x)$ if x is the root of a solid subtree, otherwise $\Delta\mathit{cost}(x) = \mathit{cost}(x) - \mathit{cost}(p(x))$, where $p(x)$ is the parent of x in the shadow tree;
- $\Delta\mathit{min}(x)$: this represents $\mathit{cost}(x) - \mathit{mincost}(x)$.

With this representation, only the root of a solid subtree is guaranteed to contain an actual value. Values in all other nodes will be represented relative to this value. Therefore, changing the value at the root (during the *addcost* operation) will implicitly change all values in the tree.

Unfortunately, the use of difference form makes some other operations more complicated. Finding the cost of a particular node, for example, now requires traversing the tree, but it can still be done in $O(\log n)$ amortized time. Another important issue with the use of difference form is that these values must be updated appropriately whenever there is rotation or splice in the ST-tree. Although it is not hard to determine how the values should be updated in this case, it is not as immediate as it would be if the values were stored explicitly, even for these simple operations.

The application for which ST-trees were developed aims to find minima over paths. One can, however, think of several other applications for which ST-trees could be used. One could, for instance, use them to compute the sum of the weights of the vertices on a path instead of picking the minimum. To support a different application, one would need to (1) define a new set of values to be stored in each node, (2) make sure these values are updated appropriately when the shadow tree changes, and (3) define rules to traverse the tree during queries. Although this can be done, it is not nearly as simple as one would hope. Sleator and Tarjan did not define a generic interface to ST-trees, which means that value updates appear interspersed with structural operations, such as pointer updates.

A natural way of defining a generic interface for ST-trees is to let the user define which values to store in each node and how to update them after each rotation or splice. This makes changes slightly simpler, but figuring out which pieces of information to keep would still be a nontrivial task, not least because values may have to be kept in difference form.

2.1.4 Undirected Trees

ST-trees were devised primarily to handle directed trees with fixed roots. In this setting, all path-related queries refer to paths between some vertex and the root of its tree. In several applications, however, the root may change, or there may be no root at all: queries refer to arbitrary paths between different vertices of the tree.

To handle these more general cases, ST-trees support the *evert* operation: *evert*(v) makes v the root of its tree. It can be implemented within the same time bound as *expose*. An ST-tree with *evert* can represent free (unrooted) trees: to query an arbitrary s-t path, it suffices to call *evert*(t) followed by *expose*(s).

Note, however, that an efficient implementation of *evert* requires a slight change to the basic data structure. This operation works by reversing the entire path between the new root and the original root. This requires the ability of reversing, in constant time, all the left/right pointers of the binary tree representing a solid path. Of course, performing this operation explicitly would be too expensive. The solution is to maintain a *reverse bit* on each node. If this bit is true, the left and right children must be swapped: to access the right child, one must follow the left pointer, and vice-versa. Moreover, the reverse bit must be stored in difference form: the actual value of the reverse bit of a node is the exclusive-or of the values stored on its ancestors. With this representation, reversing a path is easy: it suffices to negate the bit stored at the root of the binary tree that represents the path.

The main drawback of using the reverse bit, apart from the extra space it requires, is that it makes queries and updates, especially splays and splices, slightly more complicated.

Internal nodes and *evert*. As already mentioned, the data structures presented in [51] and [52] differ in the choice of binary search tree used to represent each solid subtree. They also differ in a more subtle way: in how they map solid paths to solid subtrees.

In [51], each vertex of the original path becomes a *leaf* of the corresponding binary tree, and internal nodes are added to aggregate information. Each of these internal nodes can be interpreted as representing both a subpath (between its leftmost descendant and its rightmost descendant) and an edge (between the rightmost descendant of its left child and the leftmost descendant of the right child). In [52], on the other hand, each vertex on the solid path becomes either a leaf or an internal node of the corresponding binary search tree, and no other nodes are added.

Both variants can be used if values are associated with vertices of the original forest. The first variant can also support edges easily, since each original edge will correspond to a particular node in the tree. To use the second variant in this case, one can store values relative to an edge (x, y) (y being the parent of x) on the node representing x. This can be done because each node has at most one parent. Supporting *evert*, however, presents a problem to this approach. When a path is reversed, the information relative to each of its edges would have to move from one of the endpoints to the other. We would have to transfer the information about each edge (x, y) on the path being reversed from x to y. This is too costly.

The obvious solution is to use the representation suggested in [51]. It can be used with splay trees with no asymptotic penalty. It does, however, roughly double the number of nodes in the tree, which may have some adverse effect on performance in practice.

2.1.5 Aggregating Information over Trees

An important feature of ST-trees is that a node does not need to access its middle children. Apart from operations that deal with the exposed path (in which middle children are irrelevant), all operations happen in a bottom-up fashion. For these operations, it suffices to have pointers from each middle child to its parent. The fact that there are no pointers from the parent to the middle children greatly simplifies the data structure, since the number of children can be arbitrarily large.

Unfortunately, this feature also limits the scope of ST-trees in their original form. In particular, several application require information to be aggregated not over paths, but over trees. In these cases, one does need access from a node to its middle children. The obvious approach in this case would be to store, with each node, a list of all of its children. Aggregating information about all children would take time proportional to the size of this list, which in turn depends on the degree of the vertex. This will be expensive, unless we can guarantee that all vertices in the original forest have bounded (constant) degree.

Although this assumption is reasonable for some applications, in general it is not true. The usual solution to this problem is to use *ternarization*: whenever the input has a high-degree vertex, we replace it by a chain of degree-three vertices.[2] A common technique is to replace each vertex with degree $k > 3$ by a path with $k - 2$ vertices: the first and the last vertices are each connected to two of the original neighbors, and each of the remaining vertices is connected to one. See Figure 2.2.

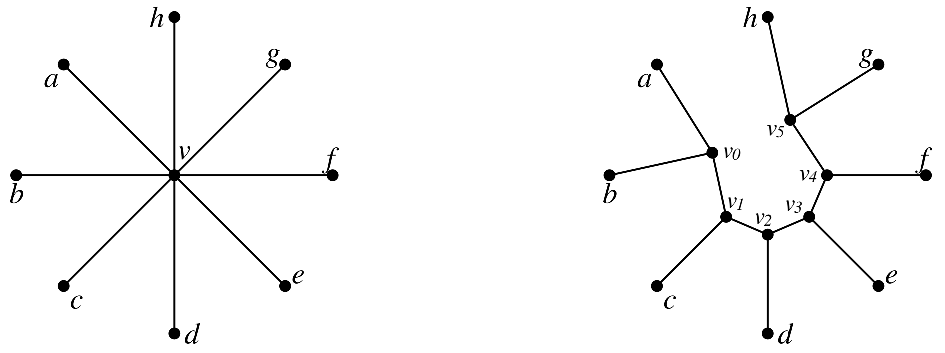


Figure 2.2: Example of ternarization. Every vertex with degree four or greater is replaced by a chain of vertices of degree three.

[2]In the case of rooted trees, by "degree-three" we mean a vertex with a parent and two children; for unrooted trees, we mean a node with three neighbors.

The drawback of this approach is that we must somehow remember which vertices and edges are in the original tree and which are special elements created by the ternarization procedure. Values associated with these extra elements typically need to be handled as special cases by the data structure.

In the papers that introduced ST-trees [51, 52], Sleator and Tarjan focused on path operations only and did not mention ternarization. As a result, the concept of ternarization has been discovered independently several times, as the following examples show.

In [27], Goldberg et al. proposed a faster implementation of Goldfarb and Hao's network simplex algorithm for the maximum flow problem [30]. The application deals with rooted trees with *labels* on vertices. In addition to the operations already supported by ST-trees, the data structure must be able to determine the vertex of minimum label among all descendants of a vertex v. To implement this, Goldberg et al. use ternarization and call the transformed tree a *phantom tree*, which is then represented as an ST-tree with some additional operations on values. They use the notion of *colors* to map vertices of the phantom tree to vertices of the original tree.

Similarly, Radzik [46] uses ternarization to implement "in-subtree" operations: (1) finding a vertex with minimum key (value) within a subtree of a rooted tree; and (2) picking a vertex at random among all that have a fixed key X in a subtree. His solution, although obtained independently, is essentially the same as the one obtained by Goldberg et al.

A third instance in which ternarization was discovered independently was within an algorithm of Langerman [41] for the so-called "shooter location problem", which is equivalent to maintaining the minimum clique cover and the maximum independent set of a circular-arc graph (a generalization of interval graphs). In order to find an efficient solution to this problem, Langerman devises a generalization of ST-trees that is a little more sophisticated than the versions above. In his setting, each node can have an arbitrary number of children, and they are *ordered*. Among the operations supported by his data structure, a contiguous subset of the children of a vertex v can be moved (in a single $O(\log n)$-time operation) to another point in the tree, becoming children of a new parent w. Once again, this can be achieved through ternarization. The set of children of a node is represented as a binary search tree, which can be split and joined with another tree when just a subset is required.

Kaplan et al. [36] apply dynamic trees with ternarization to maintain a set of intervals with priorities so that one can quickly locate the maximum-priority interval that contains a query point. Dynamic trees are used for the special case in which the intervals do not overlap (but can be nested). The authors build a *containment forest* to represent the hierarchy of intervals, and represent a "binarized" version of it using ST-trees. This technique is exactly the same as ternarization, but applied to rooted trees. As in the previous application, the order among the children of a node is relevant, and a contiguous subset of the children may acquire a new parent in a single operation.

Recently, Klein [38] devised an algorithm for finding multiple-source shortest paths on planar graphs that requires finding minimum labels over subtrees. Once again, the solution (devised independently of the results above) was to use ternarization.

We shall see that two other data structures, topology trees and RC-trees, must use ternarization as well to handle trees of arbitrary degree. In fact, unlike ST-trees, they require ternarization even if only path operations are to be supported, unless all vertices in the input are guaranteed to have bounded degree. Top trees and ET-trees, on the other hand, naturally support trees of arbitrary degree.

2.1.6 Other Extensions

Apart from ternarization, we note that ST-trees have been extended to support more complicated operations. For instance, Georgiadis et al. [26] have recently shown how one can *merge* paths. Their application considers trees that have fixed roots and are *heap ordered*: each vertex is associated with a label that must be greater than that of its parent. Operation *merge*(v, w) computes the nearest common ancestor x of v and w and merges the paths $v \cdots x$ and $w \cdots x$ so that the heap order is preserved. Even though a single operation may cause a linear number of edges to change, Georgiadis et al. show that it can be performed in $O(\log^2 n)$ amortized time with ST-trees.

2.2 Tree Contraction

We have seen that ST-trees are based on the technique of *path decomposition*: the original tree is partitioned into disjoint paths, and each is represented as a binary tree. Contraction-based data structures take a fundamentally different approach. Instead of representing the tree itself, they represent a *contraction* of the tree, which progressively transforms the original tree into smaller trees that summarize the original information.

Section 2.2.1 describes contractions in the parallel setting, where the concept first appeared. We then describe three different contraction-based data structures: topology trees (Section 2.2.2), RC-trees (Section 2.2.3) and top trees (Section 2.2.4).

2.2.1 The Parallel Setting

Contractions are based on two operations introduced by Miller and Reif [43]. The *rake* operation eliminates vertices of degree one, while *compress* eliminates vertices of degree two. They associate information with vertices. When a degree-one vertex is eliminated, the information it holds is transferred to its only neighbor (and aggregated appropriately). When a degree-two vertex v is eliminated, its two incident edges (u, v) and (v, w) are replaced by a single edge (u, w). The information associated with v is transferred to u, w, or both. A *contraction* is a sequence of *rake* and *compress* moves that reduces the original tree to a single vertex, which will hold information about the entire tree.

Miller and Reif proposed *rake* and *compress* in the context of parallel algorithms, with each vertex maintained by a different processor. They assume that the underlying tree is rooted. Their contraction algorithm works in rounds. In each round, the *rake* operation eliminates all leaves of the tree. Simultaneously, the *compress* operation will eliminate a subset of the vertices of degree two. More precisely, let a *chain* $v_1, \ldots, v_k$ be a sequence of $k \geq 2$ vertices such that v_{i+1} is the only child of v_i (for $1 \leq i < k$) and v_k has exactly one child, which is not a leaf. In each chain, the *compress* operation works by identifying vertex v_i with v_{i+1}, for all odd values of $i < k$. Intuitively, v_i accumulates the value and inherits the descendants of v_{i+1}.

Miller and Reif show that a constant fraction of the vertices is guaranteed to disappear after each round, which implies that the total number of rounds is bounded by $O(\log n)$. Since each round can be performed in constant time when there is one processor per vertex, the total running time of the algorithm is $O(\log n)$ as well.

The main application of tree contraction studied by Miller and Reif is the evaluation of expression trees. Each leaf of the original tree is associated with a real value, and each internal node with an operation ($+$ or $\times$) to be performed on its children (assumed to be exactly two). In each subsequent round, the remaining leaves will still contain a value, and each internal node with two children will be associated with an operation. Internal nodes with a single child will be associated with *linear functions* of the form $aX + b$, where a and b are constants and X is a variable representing the (not yet computed) value of the remaining child. When a node is eliminated by *rake* or *compress*, the linear expression on the parent is updated accordingly. After all rounds are completed, the only remaining vertex contains the value of the entire expression.

As observed by Miller and Reif, the *rake* operation is enough to ensure a correct evaluation of the expression. If the original tree is very unbalanced, however, up to $\Theta(n)$ rounds may be needed to contract it. This is why *compress* is also necessary. One could also think of using only *compress*, but that would not even guarantee that the whole tree would be contracted. The idea of tree contraction, with variants of *rake* and *compress*, has been further studied in the parallel setting by Abrahamson et al. [1] and by Cole and Vishkin [17].

The subsections that follow describe three contraction-based data structures, all of which use variants of *rake* and *compress*. First, we discuss Frederickson's *topology trees* [23, 24, 25]. As in the Miller and Reif setting, information is associated with vertices. We then briefly describe RC-trees, proposed by Acar et al. [3], which can be seen as a randomized version of topology trees. We shall see that storing information on vertices limits the applicability of these two data structures to trees with bounded vertex degrees, unless ternarization is used. The third data structure we discuss, Alstrup et al.'s *top trees* [10], solves this problem by storing information on *edges* instead.

2.2.2 Topology Trees

The first data structure based on tree contraction was Frederickson's *topology trees* [23, 24, 25]. Given an underlying tree in which each vertex has no more than three neighbors, the data structure represents a *multilevel partition* of the tree. The vertices of the original tree are partitioned into *clusters*, each of which contains either a single vertex or a collection of connected vertices. After each cluster is contracted into a single vertex, the result will be another tree, which can be itself partitioned into clusters. This process continues until there is only one cluster left, which will represent the entire tree.

The topology tree is merely a static representation of the hierarchy of clusters. The original tree is considered to be level zero of the contraction, with each vertex represented as an individual cluster. These clusters will be the leaves of the topology tree. In general, a cluster at level $\ell + 1$ is created by combining one or more clusters from level ℓ. The new cluster will appear in the topology tree as the parent of the clusters it contains. Each level of the topology tree can be interpreted as a contracted version of the original tree. Figure 2.3 shows a multilevel partition of the tree and a corresponding topology tree.

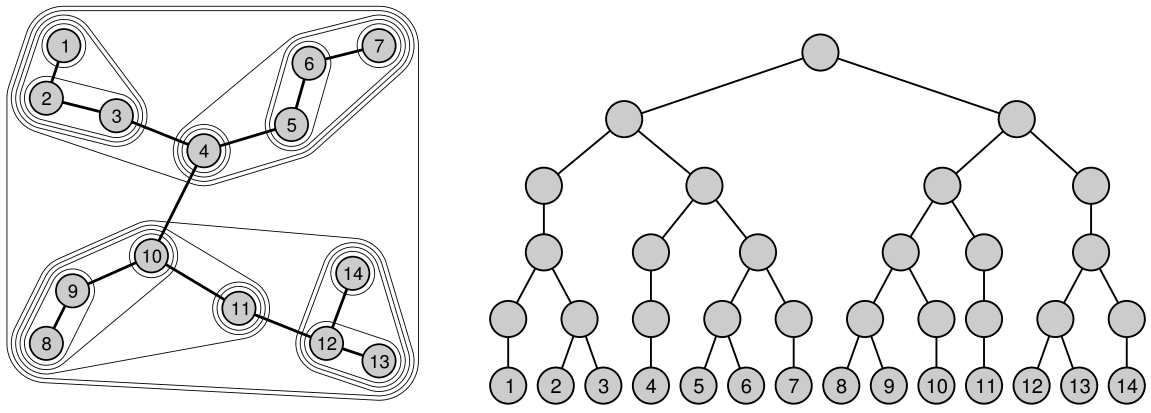


Figure 2.3: Example of a topology tree (adapted from [24]). On the left, the original tree and a multilevel partition; on the right, a topology tree that represents this partition. Pointers between neighboring clusters at the same level are not shown.

Frederickson presented at least three different versions of topology trees [23, 24, 25]. All three work in rounds, and they require all clusters formed in a round to be independent (i.e., no original vertex may belong to more than one cluster) and also that no vertex be left in a cluster by itself if it can legally belong to a composite cluster (i.e., a cluster with more than one vertex). The three versions of topology trees differ basically in the rules for creating composite clusters.

In [24], each cluster must have external degree at most three, i.e., it must be adjacent to at most three other clusters. Furthermore, if it does have external degree three, then it must contain a single vertex. The remaining clusters may have up to two vertices each. This means that a two-vertex cluster will be either (a) a degree-one vertex matched with its neighbor; or (2) two neighboring degree-two vertices matched together. Frederickson observes that a cluster of type (1) can be seen as representing a *rake* move, and those of type (2) are related to *compress*. The multilevel partition in Figure 2.3 obeys the rules presented in [24].

The version of topology trees discussed in [25] is very similar but assumes the underlying trees are rooted and binary. This still means that each node will have no more than three neighbors: the parent and two children. The original description of topology trees [23] is slightly more complicated, and mapping the rules proposed there to *rakes* and *compresses* is not immediate. Our focus in this section is on the newer, simplified versions, presented in [24] and [25].

In all versions, Frederickson shows that the number of vertices in each level is reduced by at least 1/6, which means that the height of the topology tree is $O(\log n)$.

Whenever there is a *link* or *cut*, the contraction must be updated. Frederickson proposes a bottom-up algorithm to do this. It processes each level in turn, deleting clusters that are no longer valid and inserting new ones to replace them. When processing a level, the algorithm must find out which clusters of the next level will be affected. In order to do this efficiently, topology trees must maintain not only *vertical* pointers (i.e., pointers between parents and children) but *horizontal* pointers as well (i.e., pointers between neighboring clusters on the same level).

Frederickson shows that the update algorithm can be performed in $O(\log n)$ time in the worst case. After a *link*, the list of clusters that must be inserted at any given level of the topology tree actually forms a connected subtree of the tree at that level. Moreover, there are at most two so-called *border edges*, i.e., edges between this subtree and the remainder of the tree. If the subtree were isolated, its size should decrease by at least 1/6 after each round. Because of the border edges, the actual decrease may be smaller, but only by a small additive constant factor. Frederickson shows that at most 14 new clusters may be inserted at each level after a *link*.[3] The number of original clusters that may be deleted is also bounded by a constant (although it is not specified in either [24] or [25]). According to the author, the analysis of *cut* is similar, and equivalent (unspecified) bounds can be found.

Basic applications. In [25], which deals with rooted trees, two main applications are considered: dynamic expression trees and network flows. The former aggregates information over the entire tree, while the latter does so over individual paths.

Dynamic expression trees are maintained as in Miller and Reif's original application. A cluster representing an original vertex of the tree will contain either a value or an operation ($+$ or $\times$), while other clusters aggregate information in the form of linear functions.

A more interesting application is network flows. Frederickson was the first to show that a contraction-based data structure could support the set of operations for which ST-trees were designed. We have already seen them in Section 2.1: *findcost*, *findroot*, *findmin*, *addcost*, and *parent*.

We have seen that the leaves of the topology trees correspond to the vertices of the original tree and that each internal node (cluster) of the topology tree can be seen as representing a subset of the original vertices (the ones represented by the leaves that descend from the node). Frederickson shows how one can also associate a cluster with so-called *restricted paths*, which are upward paths in the original tree. A cluster holds information about minima over these paths. As in ST-trees, these values are not stored explicitly, since *addcost* would require too many updates to the tree. They are stored in difference form.

[3]In [24], Frederickson claims that a more careful analysis might reduce both constants (1/6 and 14).

Unfortunately, updating these values is even more complicated than in ST-trees, since the mapping between clusters and paths is not as natural—there is no correspondence between binary trees and paths.

Dynamic graphs. It should be noted, however, that topology trees were originally devised to handle dynamic graph applications. In [23], topology trees are used in the solution of the *dynamic minimum spanning tree* problem. The problem is to maintain the minimum spanning tree of a graph subject to edge insertions and deletions. The simplest solution is to recompute the entire spanning tree after an insertion or deletion occurs. Frederickson demonstrated that, with topology trees, updates can actually be supported in sublinear time. More precisely, he showed that an update can be supported in $O(\sqrt{m}\log n)$ time, where m and n denote the current number of edges and vertices in the graph.

Handling insertions is relatively simple: when a new edge (v, w) is inserted into the graph, all one has to do is check if it costs less than the most expensive edge on the path between v and w on the current minimum spanning tree. This can be done with a variant of the *findmin* operation of ST-trees or topology trees. Deletions, however, are more complicated. If (v, w) is removed from the graph and it does belong to the tree, one must find a replacement edge among all edges between the component containing v and the one containing w. Frederickson uses topology trees to accomplish this. The algorithm itself is somewhat involved, but the basic idea is to make each cluster C at level ℓ remember the minimum edge between C and each of the other clusters at the same level.

2.2.3 RC-Trees

The most recent contraction-based data structure is RC-trees, proposed by Acar et al. [3, 4]. They closely resemble the original contraction scheme by Miller and Reif; in fact, "RC" stands for "rake and compress."

As in topology trees, clusters are vertices, which means that the data structure can only support trees with vertices of bounded degree. RC-trees are, however, slightly more general than topology trees. While the latter requires each vertex to have degree at most three, RC-trees naturally support any fixed upper bound d. The total space usage and the running time are proportional to d, however. RC-trees assume the original tree to be free (unrooted).

RC-trees also work in rounds, but they have different contraction rules. First of all, the algorithm strictly alternates between *rake rounds* and *compress rounds*. During a *rake* round, all degree-one vertices are removed. The contribution of each eliminated vertex is stored in its only neighbor. During a *compress* round, a subset of the vertices of degree two disappears. The two edges incident to an eliminated vertex are replaced by a single edge between the original neighbors. The neighbors are responsible for the information stored in the disappearing vertex. The algorithm requires the *compress* moves to be independent, i.e., if a vertex disappears, both of its neighbors do not. Acar et al. use randomization to ensure independence. A hash function $\mathcal{H}$ is used to map a vertex and a level to a boolean value. A vertex v on level ℓ with neighbors u and w will be *compressed* only if all of the following conditions hold: v has degree two, $\mathcal{H}(v, \ell) = \textit{true}$, and $\mathcal{H}(u, \ell) = \mathcal{H}(w, \ell) = \textit{false}$.

The height of an RC-tree is logarithmic in the number of vertices in the underlying tree, but, because *compress* rounds are randomized, this bound only holds in the expected sense. The expected update time (after a *link* or *cut*) is also logarithmic.

Although it does not guarantee good worst-case performance, randomization is useful in other respects. First, it simplifies the updating algorithm, executed whenever a *link* or *cut* occurs. The algorithm works in a bottom-up fashion, as does the updating algorithm for topology trees, but new moves can be performed independently of the original ones. A second (and related) useful aspect of randomization is that it simplifies the proof that the update algorithm takes constant time per level. As with topology trees, the proof is based on the fact that the set of new clusters on each level forms a subtree that "touches" the rest of the tree in a constant number of places. Finally, randomization makes the data structure *history-independent*, i.e., its current state depends only on the current free tree being represented. In all other data structures for dynamic trees, the current state also depends on which operations happened before, and on the order in which they occurred.

Although this is irrelevant for most applications, it may be useful in contexts where privacy or security are important [3].

Handling values. Unlike topology trees, RC-trees are "programmable," i.e., the user can easily define which problem must be solved by defining what pieces of information should be stored on each node and setting up a few functions to handle them. The update algorithm will simply call these functions whenever it decides a vertex should be eliminated. As shown in [4], three of these *call-back functions* are defined: *rake_data* (which computes the contribution of a *raked* vertex and stores it on its neighbor), *compress_data* (which computes the contribution of a *compressed* vertex and stores it on both neighbors), and *finalize_data* (which aggregates information on the final, root cluster). In addition, the user must define a *query algorithm* to traverse the tree and compute the appropriate information.

Although this interface makes this data structure much easier to use than topology trees, it still has some drawbacks. Perhaps the most important is the fact that the interface assumes that the tree has bounded degree. To represent a tree T of arbitrarily high degree, we must first ternarize it, creating a tree T'. The interface will be to T' (which has several "special" vertices and edges created by the ternarization procedure), not to T, which would be more natural. Moreover, the query algorithm tends to be as complicated as those for ST-trees and topology trees, since values must still be represented in difference form.

2.2.4 Top Trees

We now consider *top trees*, proposed by Alstrup et al. [10] (see also [7, 9]). While still based on *rake* and *compress*, they are more general than the other contraction-based data structures discussed so far, since they have no degree constraints. In addition, we shall see that their interface is also the easiest to use.

Before describing top trees in detail, we must note that the notation used here differs slightly from the one used by Alstrup et al., as do some of the assumptions we make about the tree being represented. The essential ideas, however, are the same. A more detailed discussion of the differences will be presented at the end of this section.

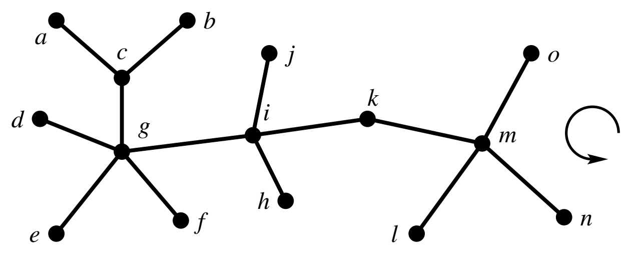


Figure 2.4: A free tree. Edges are arranged in counterclockwise order around each vertex.

Top trees represent a collection of free trees (i.e., trees that are unrooted and undirected). Furthermore, we assume there is a *circular order* of edges around every vertex of the original tree. If the application naturally has such an order, the data structure will use it; otherwise, an arbitrary order will be defined. See Figure 2.4. In this picture, as in all other representations of free trees in this work, we assume that edges are organized in **counterclockwise** order around each vertex.

In this setting, we interpret *compress* and *rake* as follows. A degree-two vertex v is *compressed* if the two edges incident to it, (u, v) and (v, w), are replaced by a single edge (u, w). We also say that either edge is *compressed at* v. A degree-one vertex v with neighbor x is *raked* if the edge (v, x) and its successor (w, x) around x are replaced by a single edge, also with endpoints w and x. We can also say that the edge (v, x) is *raked onto* (w, x), that (v, x) is *raked around* x, that (w, x) *receives* (v, x), or that (w, x) is the *target* of a *rake*. See Figure 2.5.

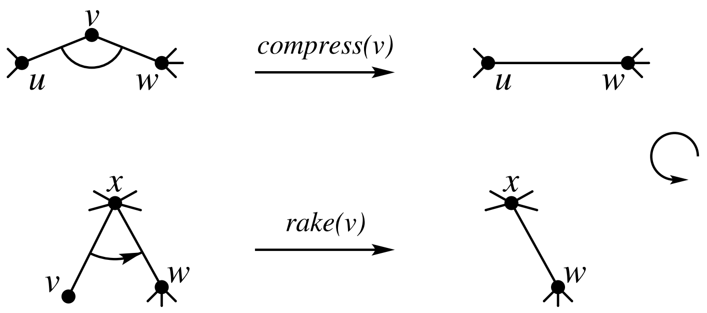


Figure 2.5: Top trees: Basic operations.

Both *rake* and *compress* are viewed as manipulating *clusters*. Each edge of the original tree is considered a cluster by itself (a *base cluster*). When two clusters are combined by either *rake* or *compress*, the result will be a new cluster, the *parent cluster.* Every cluster (not only base clusters) will have exactly two endpoints, and can therefore be considered to be an edge.

A *top tree* is a binary tree that embodies a contraction of a free tree into a single cluster via a sequence of *rake* and *compress* operations. Each leaf of the top tree is a *base cluster* representing an original edge, and each internal node is either a *rake cluster* or a *compress cluster.* A node aggregates information pertaining to all descendants; in particular, the entire original tree is represented at the root of the top tree. Since clusters correspond to edges, a tree with a single vertex will be represented by an empty top tree.

When an edge is deleted from or inserted into the original forest, there is no need to recompute a new contraction from scratch: it is enough to update the affected top trees to make them consistent with the new underlying forest. Since changes to the leaves propagate to the root, only sequences of *rakes* and *compresses* that produce *balanced* top trees can provide an $O(\log n)$ worst-case solution to the dynamic trees problem.

One such sequence can be obtained as follows: work in rounds, and in each round perform a maximal set of independent moves (each cluster participates in at most one move, and no valid move is left undone). Figure 2.6 shows a contraction of the tree in Figure 2.4 that obeys these rules; it also shows the corresponding top tree.

The top tree in the example has four types of nodes. A *rake node*, shown as a circle, represents a *rake* of its left child onto the right child. A *compress node*, shown as a square with two children, represents the *compress* of its two children (which can appear in any order). A *dummy node*, shown as a square with a single child, represents an edge that was not involved in any move in the previous round. Finally, a *base node*, shown as a square with no children, represents an edge of the original free tree. Note that dummy nodes could in principle be eliminated, as shown in Figure 2.7.

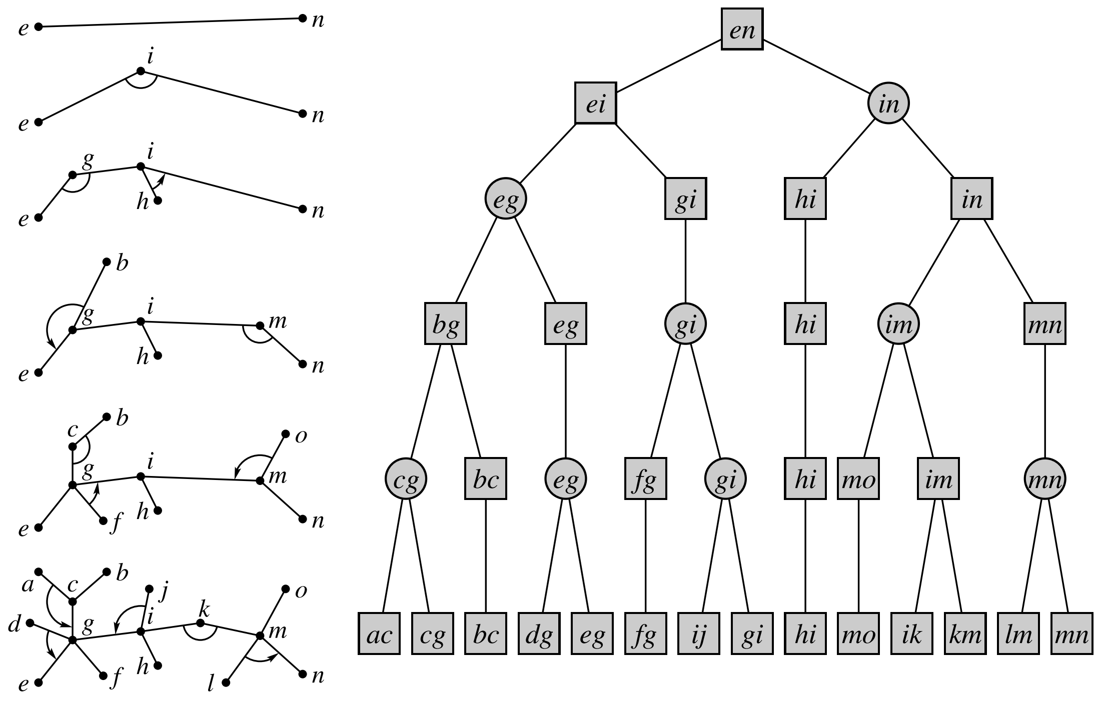


Figure 2.6: A contraction (on the left, to be read bottom-up) and the corresponding top tree (on the right). Circles represent *rake* nodes in the top tree; squares represent base nodes, dummy nodes, or *compress* nodes, depending on whether the number of children is zero, one, or two.

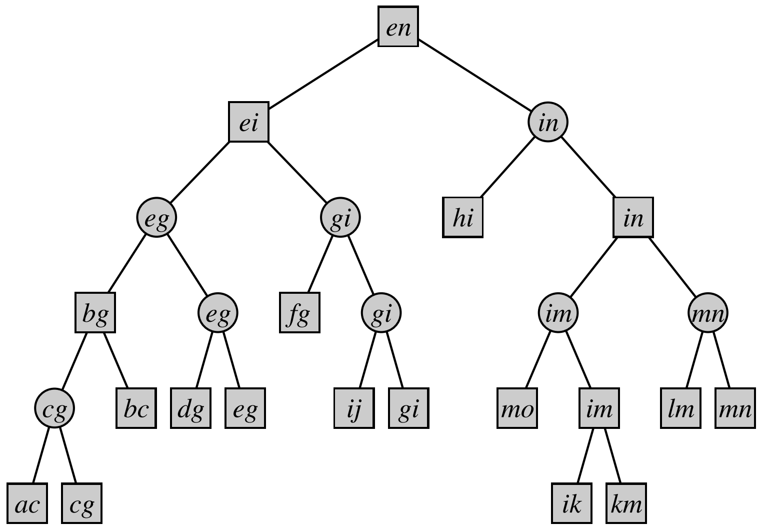


Figure 2.7: The top tree corresponding to the contraction in Figure 2.6, without dummy nodes.

Updates. Although it is fairly simple to show that this round-based contraction scheme does have at most $O(\log n)$ levels, it is not obvious that it can be updated (after a *link* or *cut*) in $O(\log n)$ time. In fact, Holm and de Lichtenberg attempted to prove this in their joint master's thesis [34] but later found the proof to be flawed.

In [10] (see also [7]), Alstrup et al. do show that top trees operations can be implemented in $O(\log n)$ worst-case time, but only as a layer on top of topology trees. Recall that topology trees are defined for ternary trees. At each level, topology clusters partition the tree into vertex-disjoint subtrees linked by *boundary edges*, which do not belong to any cluster. Vertices incident to boundary edges are *boundary clusters*. To represent an arbitrary free tree, one must first ternarize it. A topology tree is then built to represent the transformed tree. To create a top tree, the basic step is to transform each topology cluster into a top cluster, which can be done because a topology cluster is guaranteed to have at most two boundary vertices (i.e., vertices incident to boundary edges). This is obviously true for clusters with two or fewer boundary edges. It is also true for clusters with three incident edges, since the contraction rules for topology trees ensure that these clusters must have a single vertex. Every topology cluster is transformed into a top cluster induced by the vertices and edges it contains. As mentioned in [10], a topology cluster may create up to two top clusters, which can make the top tree have twice as many levels as the corresponding topology tree.

We will present in Chapter 3 a very simple update algorithm that does guarantee an $O(\log n)$ update time for the contraction scheme used in the example (based on maximal independent rounds). Since it does not rely on topology trees, it is the first stand-alone worst-case implementation of top trees. Chapter 4 presents another stand-alone implementation of top trees, which is usually faster in practice but has only an amortized performance guarantee.

Interface. In [10], Alstrup et al. focus mainly on the top tree interface and on how it makes the specialization of top trees to different applications extremely simple. The interface consists of a small set of operations that must be defined by the user (which we will present shortly). Just as importantly, the interface imposes a strict discipline for accessing the tree. Instead of allowing the user to traverse the top tree freely, she is granted direct access only to its *root*. Even though this may appear restrictive, they show that this strategy is just as powerful as any other, and in fact greatly simplifies the implementation of various applications.

Since a cluster always has two endpoints, it can be thought of as an edge on some level of a contraction of the tree. Moreover, it can be naturally mapped to both a *path* and a *subtree* of the original tree. The path is the one between its endpoints; the subtree is the one induced by all base clusters that descend from it. Take cluster ei in Figure 2.6. It represents not only the path from e to i, but also the subtree induced by edges (a, c), (c, g), (b, c), (d, g), (e, g), (f, g), (i, j), and (g, i). The vertices that belong to the subtree but are not endpoints are said to be *internal* to the cluster: the internal vertices of cluster ei are a, b, c, d, f, g, and j.

With that in mind, top trees support three basic *external operations*, which the user can call directly:

- $C \leftarrow \mathit{link}(e)$: adds an edge $e = (v, w)$ to the forest and returns the base cluster representing the new edge. If the edge would form a cycle, it is not added and the function returns *null*.
- $\mathit{cut}(C)$: removes the edge represented by base cluster C from the forest.
- $C \leftarrow \mathit{expose}(v, w)$: takes at most two vertices as input and ensures that they are endpoints of the root clusters of their trees. If v and w belong to the same component, the function returns its root cluster; otherwise, it returns *null*.

The last operation is important because of the constraint that the user has direct access only to the root of a top tree. If the user is interested in some path $v \cdots w$, she must first call $\mathit{expose}(v, w)$, then look at the cluster returned. If the user wants to deal with the tree containing v, calling $\mathit{expose}(v, \cdot)$ will give her access to the root of the relevant top tree.

To execute these operations, the clusters in the affected top trees must be rearranged.

It is up to the data structure itself to decide how to carry this out. While doing so, it must update the information stored in each of the clusters affected. The exact pieces of information (and how they are updated) depend on the application the top trees are solving. Therefore, it is up to the user to define what fields each cluster should have, and how they should be updated. To that end, the user must provide the implementation of four *internal operations*, defined by Alstrup et al. as follows:

- $C \leftarrow \mathit{create}(e)$: makes a base cluster representing edge e;
- $C \leftarrow \mathit{join}(A, B)$: performs a *rake* or *compress* of two adjacent clusters;
- $(A, B) \leftarrow \mathit{split}(C)$: disassembles a *rake* or *compress* cluster and returns its children;
- $\mathit{destroy}(C)$: eliminates a base cluster.

When dummy nodes are present, both *join* and *split* must be generalized to allow clusters with a single child. Moreover, both functions should have an additional parameter specifying the type of move to be made (*rake*, *compress*, or dummy). Following Alstrup et al., we have not explicitly included this parameter in the definitions above to simplify notation. If the move is a *rake*, *join* assumes that the first cluster (A) is to be *raked* onto the second (B); *split* adopts the same convention for its output.

The four internal operations are defined as call-back functions. The data structure will transform structural changes to the tree into a series of *create*, *destroy*, *split*, and *join* operations. At most one *create* or *destroy* is executed per external operation, since only one original edge is inserted or removed at a time (in the case of *expose*, none is). On the other hand, $O(\log n)$ internal clusters may change, causing as many calls to *split* and *join*.

Each application of dynamic trees will have different pieces of information associated with each cluster, and will implement the internal operations to handle this information appropriately. The remainder of this section presents a few examples to illustrate the power and simplicity of top trees. Unless otherwise noted, the applications presented here are described in [10].

Aggregating information over trees. Perhaps the simplest possible application of top trees is to maintain the sum of the costs of all edges in a tree. In this case, we store a single value in each cluster. The *create* operation initializes this value as the cost of the corresponding edge; *join* stores in the new cluster the sum of the values in the children. Both *split* and *destroy* do nothing.

If, instead of finding the sum, we were interested in finding the minimum edge cost, we would just have to change *join*: instead of computing the sum of the values in its children, it would compute the minimum.

Operations on paths. A slightly more involved application is to find the length of a given *path* in the tree. We store a single value per cluster, corresponding to the length of the path between its two endpoints. As in the previous application, the *create* operation initializes this value as the length of the corresponding edge. The behavior of *join* depends on whether the operation being performed is *compress* or *rake*. When two edges (u, v) and (v, w) are *compressed*, *join* stores the sum of their values in the parent cluster. If (u, v) is *raked* around v onto (v, w), *join* will merely copy the value of (v, w) to the parent cluster. Note that this captures the essence of each operation: a *compress* effectively combines two paths into one, whereas a *rake* merely "discards" one of the paths and keeps the other—after all, the parent edge has the same endpoints as the receiver.

These update rules ensure that the root cluster of the top tree will contain the length of the path between its endpoints. To compute the length of the path between two arbitrary vertices v and w, all the user needs to do is call *expose*(v, w) and read the value in the cluster returned by the operation.

Maintaining diameters. The top tree interface makes it easy to implement applications that require information about subtrees *and* paths at the same time. An example is maintaining the *diameter* of a dynamic tree, i.e., the largest distance between two vertices in the tree. Using top trees, Alstrup et al. were the first to show that the diameter of a tree can be updated in $O(\log n)$ time after a *link* or *cut*. To achieve this, we maintain in each cluster

$C = (v, w)$ not only its diameter—denoted by $\mathit{diam}(C)$—but also some auxiliary fields that will allow us to update the diameter efficiently. We must store the length of the path between v and w, denoted by $\mathit{length}(C)$, and the maximum distance from each endpoint to a vertex in the subtree represented by C. This requires one field for each endpoint: $\mathit{max}_v(C)$ and $\mathit{max}_w(C)$.

Defining *create* and *join* to update these fields appropriately is quite simple. We have already seen how the *length* field can be maintained, so we restrict our discussion to the other three fields. When a new base cluster C representing an $e = (v, w)$ with length ℓ is created, operation *create* must set $\mathit{diam}(C) = \mathit{max}_v(C) = \mathit{max}_w(C) = \ell$. When two clusters A and B with a common vertex v are combined into a cluster C, *join* must set

$$\mathit{diam}(C) = \max\{\mathit{diam}(A), \mathit{diam}(B), \mathit{max}_v(A) + \mathit{max}_v(B)\}.$$

Also, let w be an endpoint w of C that belongs to B but not A. We update $\mathit{max}_w(C)$ as follows:

$$\mathit{max}_w(C) = \max\{\mathit{max}_w(B), \mathit{length}(B) + \mathit{max}_v(A)\}$$

If C is a *compress* cluster, it will also have an endpoint u that belongs to A but not B. Field $\mathit{max}_u(C)$ must be updated in a similar fashion to $\mathit{max}_w(C)$. If the move is a *rake*, v will be an endpoint of C, and we just set $\mathit{max}_v(C) = \max\{\mathit{max}_v(A), \mathit{max}_v(B)\}$.

Non-local search. In [10], Alstrup et al. observe that certain applications require performing a binary search within the top tree to find a base cluster with some specific property. These applications include maintaining the *center* of a dynamic tree (i.e., the vertex that minimizes the distance to the farthest vertex) or its *median* (the vertex that minimizes the sum of the distances to all other vertices).

The top tree interface, however, forbids the user from performing the binary search directly, since it would involve looking at non-root nodes. Instead, the authors propose a routine that gradually transforms the original top tree into another, with the target edge represented at the root. A fifth user-defined internal function, *select*, is used to guide this construction. Alstrup et al. show how *select* can be implemented independently of the other operations (*link*, *cut*, and *expose*) to run in time proportional to the original depth of the base node representing the target edge. As soon as the desired query is completed, the original tree is restored.

Implicit values. In the examples presented so far, only *create* and *join* needed to be defined. The other two internal functions, *split* and *destroy*, did nothing. They are useful when values must be represented implicitly. An obvious example is the maximum flow application that motivated ST-trees. In this case, one must not only find the edge of minimum cost along a path, but also have the ability of adding, in a single operation, a constant value x to each edge on a path.

To support these operations, we maintain two values in each cluster C. The first is *mincost*(C), the cost of the minimum edge on the path between the endpoints of C. The second value is *extra*(C): this is a "lazy value" to be added to all clusters that represent subpaths of C, excluding C itself.[4] These clusters are all descendants of C in the top tree. Although essentially equivalent to difference form, lazy values fit more naturally with the top tree interface and are often easier to reason about.

The internal operations are defined so that the following invariant always holds: if R is the root of the top tree, *mincost*(R) will contain the actual minimum edge cost on the path between the endpoints of R. This will not necessarily be true for other clusters in the tree.

The operations are defined as follows. Operation *create*, when applied to an edge with cost x, initializes *mincost* to x and *extra* to zero. Operation $C \leftarrow$ *join*(A, B) will always set *extra*(C) to zero. The value it sets for *mincost*(C) depends on the operation being performed: if it is *compress*, *mincost*(C) is set to the minimum of *mincost*(A) and *mincost*(B); if the operation is *rake*, *mincost*(C) is set to *mincost*(B) (recall that B is the receiver).

To add a value x to the cost of each edge from v to w, it suffices to call *expose*(v, w) and add x to both fields (*mincost* and *extra*) of the root cluster. This value will only be propagated to the rest of the tree when the cluster is split. More precisely, when *split* is applied to a *compress* cluster, its *extra* value is added to both fields (*mincost* and *extra*) of

[4]Alternatively, one could include C, as long as the internal operations are defined properly.

each children. When applied to a *rake* cluster, *split* adds the *extra* value to both fields of the child representing the receiver; the other child remains unaltered. The *destroy* operation does nothing.

Data on vertices. Since top tree clusters correspond to edges, representing edge-related information is trivial. In many applications, however, we must associate data with *vertices* instead. Alstrup et al. [10] suggest attaching to each vertex v a special edge (a *label*) to store vertex-related data; one of its endpoints is v, and the other a dummy vertex with no other incident edge. Although this approach is generic, adding extra edges is an undesirable overhead. For most applications, it is enough to keep the data associated with the vertices in a separate array to which the internal operations (*join*, *split*, *create*, and *destroy*) have access. Vertex information would be explicit for exposed vertices (i.e., those that are either isolated or endpoints of root clusters), and implicit for internal ones. Only the values of exposed vertices could be accessed directly. To query v, one would first call $\mathit{expose}(v, \cdot)$, then look at the entry for v in the array.

Suppose, for example, that we want to maintain the total weight of the vertices in a tree. An auxiliary array keeps individual weights, while each cluster stores a single value corresponding to the total weight of its internal vertices (ignoring its endpoints). Operation $\mathit{create}(e)$ initializes this value as zero, since a base cluster has no internal vertices. The *join* operation sets the value of the parent cluster to be the sum of the values in its children plus the weight of the disappearing vertex, taken from the auxiliary array. To find the total weight of the tree containing vertex v, we first call $C \leftarrow \mathit{expose}(v, \cdot)$, then return the value stored in C plus the weights of its two endpoints, available at the auxiliary array. To change the weight of a vertex v, we first expose it, then change the value in the auxiliary array.

Notation. In their presentation of top trees [10], Alstrup et al. use a different terminology from the one we adopt here. Instead of dealing with *rakes* and *compresses*, they think of *join* and *split* as manipulating *path clusters* and *point clusters*. A path cluster shares both of its endpoints with other clusters; a point cluster shares at most one. Intuitively, point clusters are those that can be *raked*; a path cluster can be *compressed* if its endpoints have small enough degree. We believe the notation using *rake* and *compress* explicitly is more natural, and it stresses the similarity between top trees and other contraction-based data structures.

Another difference between the our interface and that of Alstrup et al. is that we explicitly support sorted adjacency lists (i.e., there is a circular order around each vertex). Since we allow the order to be arbitrary, our representation is slightly more general.

2.3 Euler Tours

The third basic approach for representing dynamic trees is that used by *Euler tour trees*, or ET-trees. They were originally introduced by Henzinger and King [32, 33] to help maintain dynamic graph properties (connectivity, bipartiteness, and approximate minimum spanning trees) in polylogarithmic time per edge insertion or deletion. The data structure was later simplified by Tarjan [56]. We focus on Tarjan's version of the data structure.

Tarjan uses ET-trees to aggregate information over the vertices of a tree. Every vertex v has an associated value $val(v)$. Apart from the usual *link* and *cut* operations, the data structure must support operations that deal with values. Two operations deal with individual vertices: $findval(v)$ returns $val(v)$ and $changeval(v, x)$ sets $val(v)$ to x. Two other operations deal with information about the entire tree: $findmin(v)$ returns the vertex of minimum value in the tree containing v, and $addval(v, x)$ adds x to all vertex values in the tree containing v.

The data structure is based on a very simple idea: represent the tree by an Euler tour of its edges, and represent the tour itself as a binary tree. The Euler tour of a tree is a tour that traverses each edge of the graph twice (once in one direction, once in the other). In general, a tree may have exponentially many Euler tours. Any one of them can be used by the data structure.

Besides arcs, the Euler tour also includes nodes representing the vertices of the tree. Each vertex w corresponds to a single node in the Euler tour, and it appears right after a node representing an arc (v, w), for some v. Note that, if w has degree greater than one, there will be more than one such arc in the tree (and the tour). Any one can be chosen to be the predecessor of the node representing vertex w. See Figure 2.8.

The data structure represents a forest as a collection of Euler tours. If an edge (v, w) is *cut* from the forest, we must remove the arcs (v, w) and (w, v) from the Euler tour and patch the remaining nodes into two independent Euler tours. Conversely, if an edge (v, w) is inserted into the forest (by a *link* operation), we do the opposite: we insert arcs (v, w) and (w, v) to combine two Euler tours.

The obvious representation of the tour itself is as a doubly-linked list. This allows an arc to be inserted or deleted in constant time, given a pointer to it. Operations such as *findmin* or *addval*, however, would take linear time, since they would require traversing the entire tour.

To make these operations more efficient, ET-trees represent the tours as *binary search trees.* We first convert the tour into a linear list by breaking it at some arbitrary point, then we build a binary search tree in which the elements of the list appear in symmetric order. The binary tree in Figure 2.9 is a possible representation of the circular list in Figure 2.8.

This representation allows *links* and *cuts* to be performed in $O(\log n)$ time with a constant number of *joins* and *splits* of binary trees. In his presentation of ET-trees [56], Tarjan suggests using splay trees [52], which guarantee that each of these operations takes $O(\log n)$ amortized time. Slightly more complicated alternatives, such as red-black trees [31], guarantee a bound of $O(\log n)$ in the worst case.

Note that $O(\log n)$ time per *link* or *cut* is worse than what we would have with doubly-linked lists. But now we can perform *findmin* and *addval* in $O(\log n)$ time, much faster than before. This can be done if values are stored in difference form, as in ST-trees. Each node x in the binary tree stores two values:

- $\Delta val(x)$: the difference between the actual value of x and that of its parent in the binary tree (except if x is the root, in which case $\Delta val(x)$ represents the actual value);
- $\Delta minval(x)$: the difference between $val(x)$ and the minimum value in the subtree

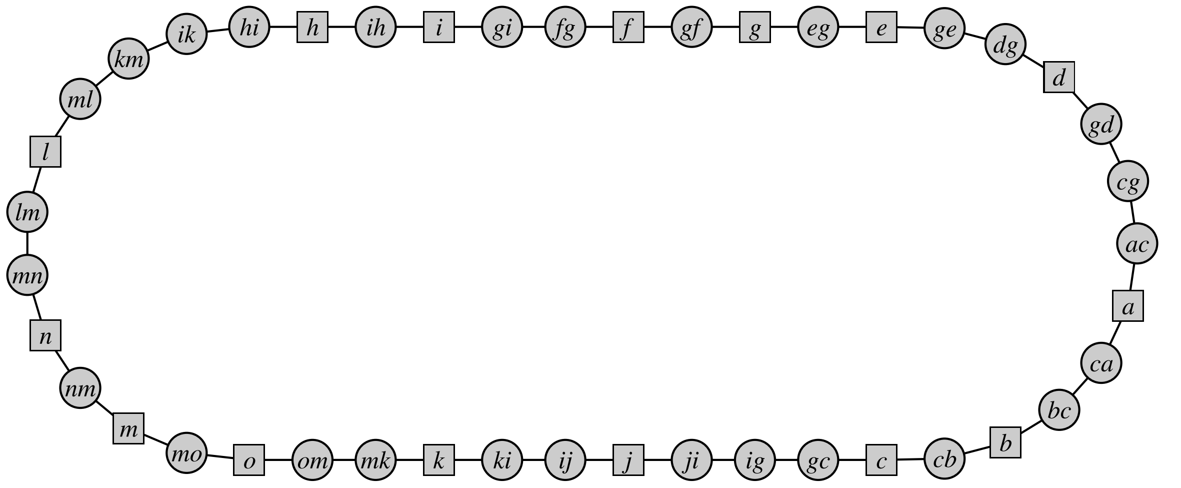


Figure 2.8: An Euler tour (as used by ET-trees) corresponding to the tree in Figure 2.4. To be read in counterclockwise order.

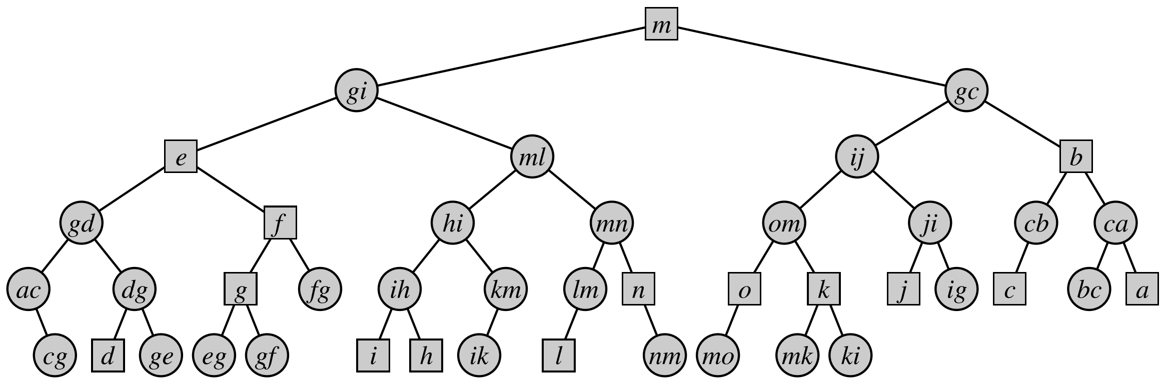


Figure 2.9: A binary search tree representing the Euler tour in Figure 2.8.

rooted at x in the binary tree.

If values are stored in this form, *addval* and *findmin* can be easily implemented in $O(\log n)$ time. Note that only nodes representing vertices of the original tree will have values associated with them. In the definitions above, a node representing an arc has no value by itself; it simply relays the information from its descendants to its ancestors. If a node representing an arc has no descendant, $\Delta val(x)$ and $\Delta minval(x)$ can be simply ignored.

ET-trees are significantly simpler than the other data structures discussed so far, which at first glance would make them the preferred choice in dynamic tree applications. The trouble with them is that they cannot handle path queries efficiently. Each edge in the forest appears as two nodes in the representation, and they may be arbitrarily far apart. This makes it hard to aggregate information about a specific path. Aggregating information over the entire tree, however, is relatively simple.

Chapter 3

Contraction-Based Top Trees

This chapter presents a very simple implementation of the top tree interface that supports *link*, *cut*, and *expose* in logarithmic time in the worst case. Our data structure uses the contraction procedure proposed by Holm and de Lichtenberg [34]. The contraction works in *rounds*. The set of moves performed in each round is *independent* and *maximal*: each edge can participate in at most one move, and no legal move is left undone. Section 3.1 will show that this strategy is guaranteed to eliminate a constant fraction of the original vertices after each iteration, which implies that the top tree will have logarithmic height.

We are interested in the *update problem*: given a contraction C of a forest F, find a contraction C' of a forest F' that differs from F in at most one edge (which has been added or deleted). The goal is to obtain C' from C with at most $O(\log n)$ modifications, where n is the total number of vertices in the trees involved.

Section 3.2 suggests a very simple greedy algorithm for updating the tree and proves that it actually achieves this goal. After a structural operation (*link* or *cut*), it simply keeps the original clusters that remain valid and greedily creates new clusters to ensure maximality. Section 3.3 details how the update algorithm can actually be implemented. Finally, Section 3.4 discusses some alternative design choices that could be made in the implementation of the data structure.

3.1 Number of Levels

A necessary—but not sufficient—condition for updating a top tree on n nodes in $O(\log n)$ time in the worst case is that its height (i.e., the number of levels of the contraction it represents) never exceeds $O(\log n)$. Before considering the update problem itself, we prove that the contraction rules we use do result in a top tree with at most $O(\log n)$ levels.

Lemma 1 *If a contraction procedure is guaranteed to eliminate a fraction $\alpha > 0$ of all nodes in each round, then the number of rounds in the contraction is at most $\log_{1/(1-\alpha)} n$.*

Proof. Let h be the round after which there are only two vertices (i.e., an edge) left. The relation

$$n(1-\alpha)^h \leq 2$$

can be derived directly from the hypothesis: we start with n vertices, and a fraction of at most $1-\alpha$ will remain after each step. Taking the binary logarithm of both sides[1] and rearranging the terms, we get

$$h \leq \frac{1-\log n}{\log(1-\alpha)} = -\frac{\log(n/2)}{\log(1-\alpha)} \leq \log_{1/(1-\alpha)} n,$$

as desired. □

Given the lemma, all we need now is a constant α that applies to the contraction rules used by our data structure. The following result will help us determine this constant.

Lemma 2 *Let D_i be the number of vertices with degree i on a free tree T. If T has at least two vertices, the following identity holds:*

$$\sum_{i=1}^{\infty} D_i(i-2) = -2. \tag{3.1}$$

Proof. Since iD_i is the total degree of all vertices of degree i and the number of edges is $n-1$, we have that

$$\sum_{i=1}^{\infty} iD_i = 2(n-1).$$

[1]In this dissertation all logarithms are base two, except when another base is given explicitly.

Replacing n, we get

$$\sum_{i=1}^{\infty} iD_i = 2\left(\sum_{i=1}^{\infty} D_i\right) - 2.$$

The lemma follows by rearranging the terms of this expression. $\square$

Lemma 3 *In any tree T, more than half of the vertices have degree one or two.*

Proof. In the summation in Equation 3.1, each degree-one vertex has a negative contribution of one unit, vertices of degree two have no contribution at all, and vertices of degree three or greater have positive contributions (of at least one unit each). Because the final result is negative (-2), there must be more vertices of degree one than of degree three or more. Since all other vertices have degree two, the lemma follows. $\square$

This is relevant to our algorithm, because every degree-one vertex is a *rake* candidate and every degree-two vertex is a *compress* candidate.

Lemma 4 *If $n \geq 3$, at least 1/6 of the vertices disappear from one round to the next.*

Proof. Let n be the number of vertices in the tree. If $n \leq 6$, the lemma is trivial: at least one move will be performed. Assume that $n > 6$. We would like to show that each move that actually happens will block at most two other potential moves (i.e., at most two other potential moves would involve edges that participate in the move that actually happens). Since there are at least $n/2$ potential moves, this would suffice to show that at least $n/6$ moves will happen. We shall see that this is not actually true: there are cases in which a move may block three other potential moves. These cases can be handled separately, however.

Consider what happens when an edge (v, w) is *raked* around v onto some edge (v, x). If v has degree two, only two moves can be blocked: *compress*(v) and a move around x involving (v, x) (which could be either *compress*(x) or *rake* with (v, x) as a receiver).

If v has degree greater than two, the following moves may be blocked: (i) a *rake* around v with (v, w) as a receiver; (ii) a *rake* of (v, x) around v; (iii) a move around x involving

(v, x) (which could be either a *rake* with (v, x) as a receiver or *compress*(x)). These are three cases, but case (ii) requires x to have degree one, and case (iii) can only happen if degree of x is greater than one. Therefore, at most two of these cases can actually happen simultaneously. Note, however, that case (iii) may actually represent two moves (both a *rake* and a *compress*), as long as x has degree two and its other neighbor (besides x) has degree one. This corresponds to configuration (a) in Figure 3.1.

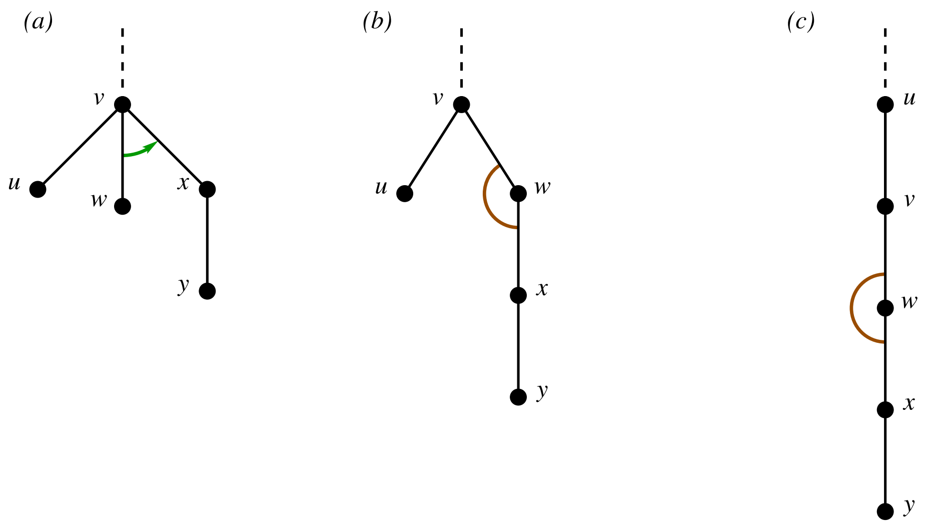


Figure 3.1: The three bad configurations, in which a single move (*rake*(w) in configuration (a), *compress*(w) in the other two configurations) can block three other moves. The vertex attached to the rest of the tree (v in the first two configurations, u in the third) may have arbitrarily high degree. The remaining vertices have exactly the degrees shown in the picture.

Now consider what happens when a vertex w with neighbors v and x is *compressed*. It may block the following moves: (i) *compress*(v); (ii) a *rake* around v with with (v, w) as the receiver; (iii) *compress*(x); and (iv) a *rake* around x with (w, x) as the receiver. For all four moves to be candidates, v, w and x must have degree two and the other neighbors of v and x (besides w) must have degree one. Therefore, the tree must actually be a 5-vertex path. Since we are assuming that $n > 6$, at most three of these moves may actually be candidates. For exactly three to be candidates, without loss of generality, x must have degree two and y must have degree one, where y is x's other neighbor besides w. This corresponds to configurations (b) and (c) in Figure 3.1.

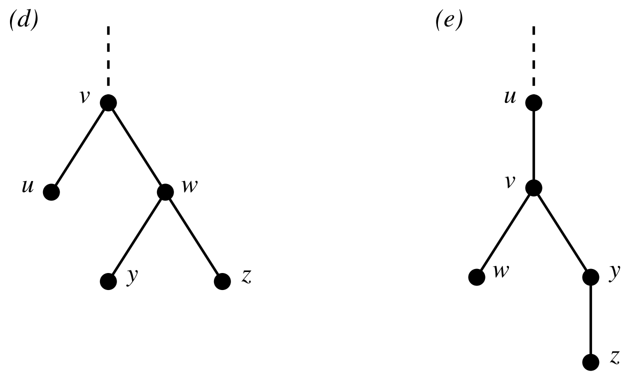


Figure 3.2: Two *good configurations* that must replace the bad configurations in Figure 3.1 in the proof of Lemma 4. Configuration (d) replaces (a) or (b), and (d) replaces (c).

We say that the configurations in Figure 3.1 are *bad.* If these configurations are absent from the tree in a given round, it is easy to show that the lemma holds, since each move actually executed will block at most two other moves. When these configurations are present, however, we need to be more careful. We will show that, when k bad configurations occur in the original free tree T, the number of potential moves is guaranteed to be at least $n/2+k$. This is large enough to account for the extra moves blocked by bad configurations.

More precisely, consider what happens when we replace each of the k bad configurations in T with one of the alternative configurations depicted in Figure 3.2 (which we call *good*). Each bad configuration of type (a) or (b) must be replaced by a good configuration of type (d); a bad configuration of type (c) must be replaced by a good configuration of type (e). Let T' be the tree thus obtained.

Now consider the properties of these configurations. First, each good configuration has the same number of vertices as the bad configuration it replaces. Second, the degree of the boundary vertex (the only vertex adjacent to the remainder of the tree) in configuration (d) is always greater than two, as it is in the bad configurations it replaces; in configuration (e), the boundary vertex will have the same degree as in the original tree, which is at least two. Third, each good configuration has exactly three non-boundary vertices of degree one or two; each bad configuration, on the other hand, has four such vertices.

Both T and T' have n vertices. Lemma 3 ensures that at least $n/2$ of the vertices in T' have degree one or two. Together with the properties above, this ensures that T has at least $n/2 + k$ vertices of degree one or two.

Let a *bad move* be a move that blocks three other potential moves; all other moves are *good moves*. There is at most one bad move per bad configuration. Consider a maximal set of moves in T. Each good move will block at most two other moves; each bad move will block three other moves, but there can be at most k of those. Because there are $n/2 + k$ potential moves in T, a maximal set of actual moves must have cardinality at least $n/6$. This completes the proof. $\square$

This establishes a lower bound on the number of edges eliminated. This is enough to bound the height of the top tree.

Lemma 5 *Any top tree representing a tree on n vertices using a maximal set of independent moves on each level will have height at most $3.802 \log n$.*

Proof. Together, Lemmas 1 and 4 ensure that the height will be at most $\log_{6/5} n$, which is less than $3.802 \log n$. $\square$

The next lemma shows that the bound given by Lemma 4 is tight up to a small additive factor.

Lemma 6 *There exists a family of trees such that a tree with n vertices has a maximal set of independent valid moves that eliminates no more than $n/6 + O(1)$ of its vertices.*

Proof. Consider a tree formed by a root r with three neighbors, each of which is itself the root of a complete binary tree of size $k = 2^c - 1$, for some positive integer c. This graph has $3k+1$ vertices, $3(k+1)/2$ of which are leaves; all other vertices have degree three. Now replace each degree-one vertex v by configuration (b) depicted in Figure 3.1. (One could also replace them with configuration (a).) See Figure 3.3.

The transformed tree will have $4[3(k+1)/2] = 6(k+1)$ more vertices than the original tree. The total number of vertices will therefore be $n = 3k + 1 + 6(k+1) = 9k + 7$. A maximal set of moves can be obtained by executing *compress*(w) in each copy of configuration (b), where w is the degree-two node adjacent to the original vertex v, as shown in Figure 3.1. The total number of moves performed in this case is $3(k+1)/2 = 1.5k + 1.5$ (one per original leaf). Note that $1.5k + 1.5 = (9k+7)/6 + 1/3$, which means that exactly $n/6 + 1/3$ vertices will be eliminated from this n-vertex tree. □

Note that this worst-case example has a very restricted structure. The author has not been able to find an example where 1/6 of all vertices are eliminated in two or more consecutive rounds, even ignoring additive constant factors. A particularly bad case in the long run (across more than one round) is that of three complete binary trees with the same number of vertices and whose roots are all adjacent to a common vertex, as shown in Figure 3.4. Ignoring additive terms, only 1/4 of the vertices (half of the leaves) are eliminated in the first round. In the following round, 1/3 of the vertices will be removed: every leaf edge will be combined with its neighbor, either by *rake* or *compress*. The result will be once again a set of three complete binary trees. Therefore, the algorithm will alternate between removing 1/4 and 1/3 of the vertices, which amounts to removing 1/2 of all vertices after each pair of rounds.

3.2 Updating the Contraction

We have proved that any contraction that works in maximal independent rounds will have at most $O(\log n)$ levels, where n is the number of vertices in the forest. Although necessary, this condition is not sufficient for our purposes. We must prove that we can *update* any such contraction in $O(\log n)$ time after a *link* or *cut*. This requires "repairing" the original contraction by undoing some of the existing moves and performing new ones. Our goal is

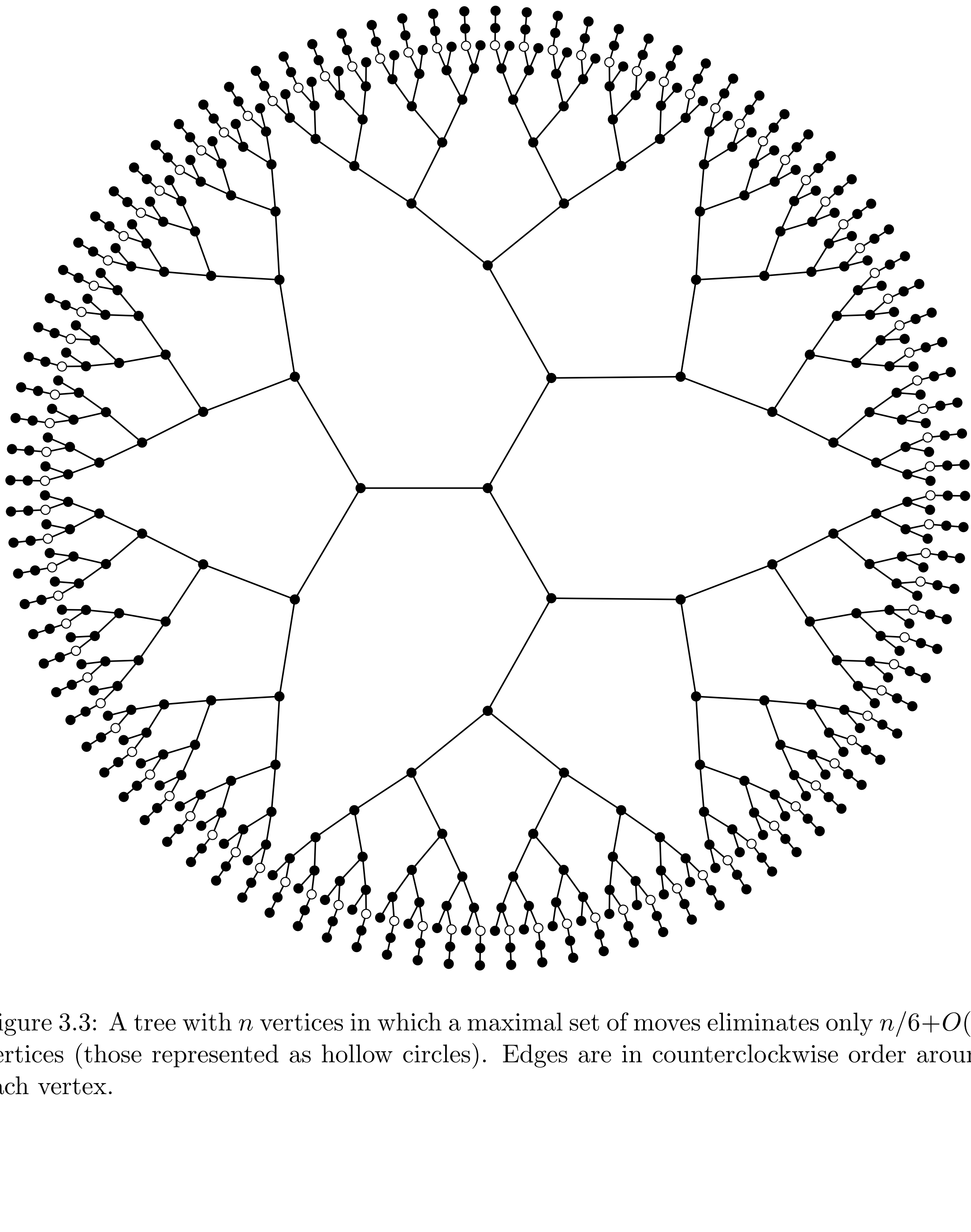
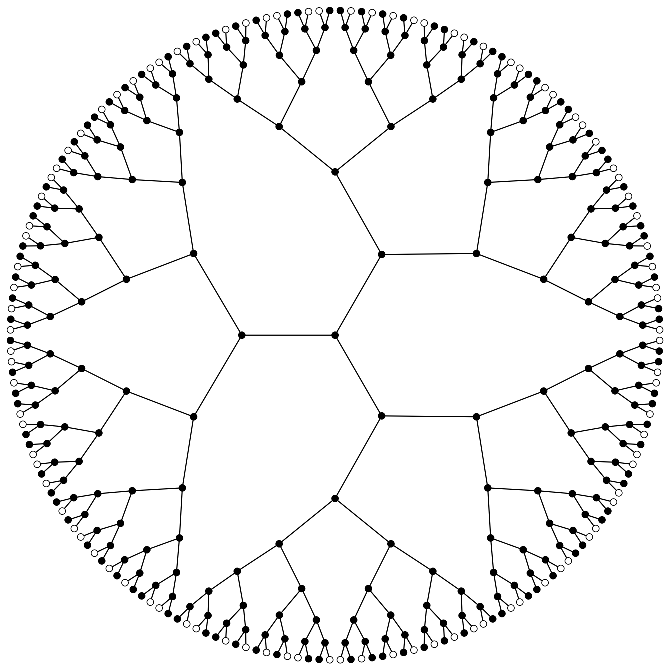


Figure 3.3: A tree with n vertices in which a maximal set of moves eliminates only $n/6+O(1)$ vertices (those represented as hollow circles). Edges are in counterclockwise order around each vertex.



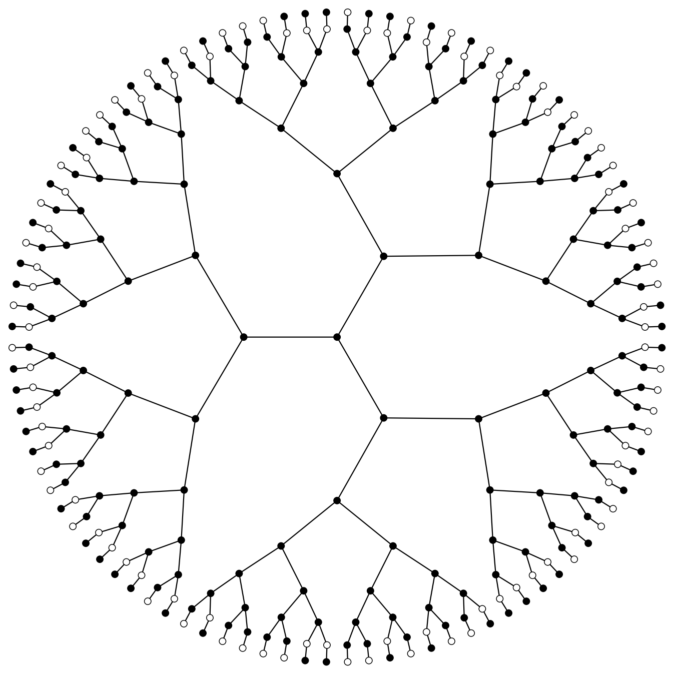


Figure 3.4: A family of trees that requires two rounds to eliminate half of all vertices. Starting from the tree on top, a maximal set of moves will eliminate 1/4 of all vertices. The elements of one such maximal set are shown as hollow circles. The resulting tree will be the one at the bottom; a maximal set of moves will eliminate 1/3 of the remaining vertices. The resulting tree will be similar to the one on top, but with one fewer level and half as many vertices.

to prove that this can indeed be carried out in $O(\log n)$ time.

When describing the algorithm, we will use the terms *original contraction* and *new contraction* to refer to the initial forest and to the forest after the *link* or *cut* operation, respectively. The update algorithm transforms the original contraction into the new one. To make the discussion simpler, however, we will describe the algorithm as if it created the new contraction from scratch. This involves saying that some moves will be *replicated*, which just means that the original cluster representing the move will be preserved.

The rules for repairing the contraction are very simple. For each level (starting from the bottommost), we just replicate all original moves that can be replicated, then perform new moves until maximality is achieved. There is no additional constraint on the set of new moves: any maximal set is a valid choice. Figure 3.5 shows an example of how an original contraction may be updated after a *link*.

The first step of the algorithm (replicating original moves) is done implicitly—we just keep the original clusters. Only the second step—performing new moves—is done explicitly. In this section, we shall prove that (1) the number of new clusters per level is constant after a *link* or *cut*[2] and (2) these clusters can be processed in constant time. To do this, we need some additional concepts.

3.2.1 Updates: Basic Notions

Active clusters. A base cluster is *inactive* if it appears in both contractions. A *rake* or *compress* (or dummy) cluster is inactive if it appears in both contractions and has the same children (both inactive). All other clusters are *active*. Note that the active clusters in the new contraction are the clusters created by the update algorithm; only on those do we need to call *join* (or, in the case of a base cluster, *create*). In Figure 3.5, active clusters in the new contraction are represented as thicker edges. Active clusters in the original contraction are those that must be deleted; we must call *split* on them (or, in the case of a base cluster, *destroy*).

[2]This is also true for *expose*, but we will only deal with it in Section 3.3.5.

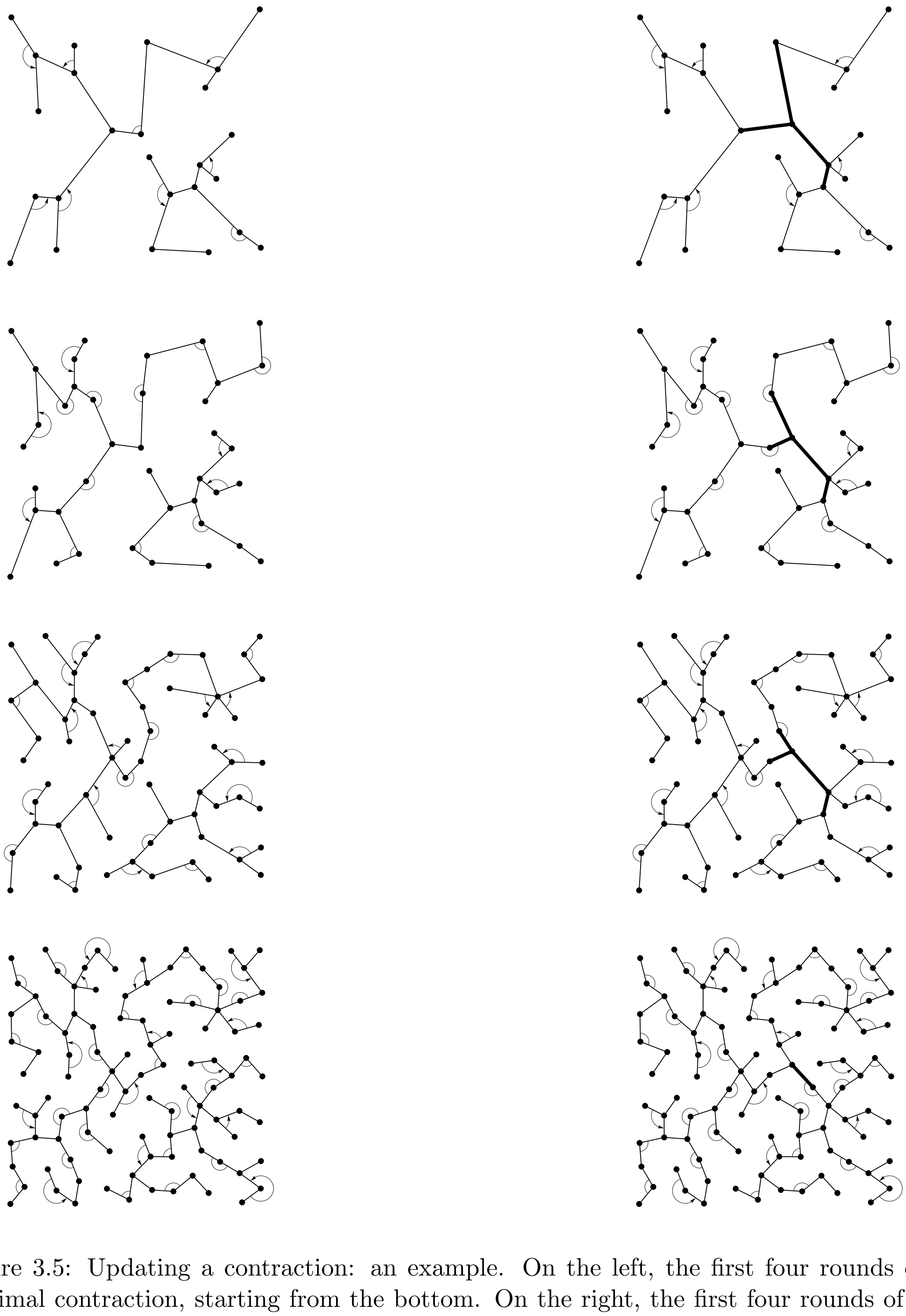


Figure 3.5: Updating a contraction: an example. On the left, the first four rounds of a maximal contraction, starting from the bottom. On the right, the first four rounds of the updated contraction after a *link* operation. The active (new) clusters in the new contraction are highlighted. Note that all original moves that can be replicated are replicated.

Consider a fixed level ℓ in the new contraction. We call the subgraph induced by the active edges the *core* of this level. The subgraph induced by the active edges on the corresponding level in the original contraction is the *core image*.

Euler tours. The existence of a circular order around each vertex (i.e., the fact that the adjacency lists are sorted) defines a unique Euler tour of each tree in the forest. For our purposes, an Euler tour of a tree is defined as a circular list of arcs. Each original (undirected) edge (u, v) will be represented as two (directed) arcs in the Euler tour: (u, v) and (v, u). We say that these arcs are *twins*. If edge (v, w) succeeds edge (v, u) in the adjacency list of v, then arc (v, w) will be the immediate successor of arc (u, v) in the Euler tour. In particular, if v has degree one, the successor of arc (u, v) will be arc (v, u). See Figure 3.6.

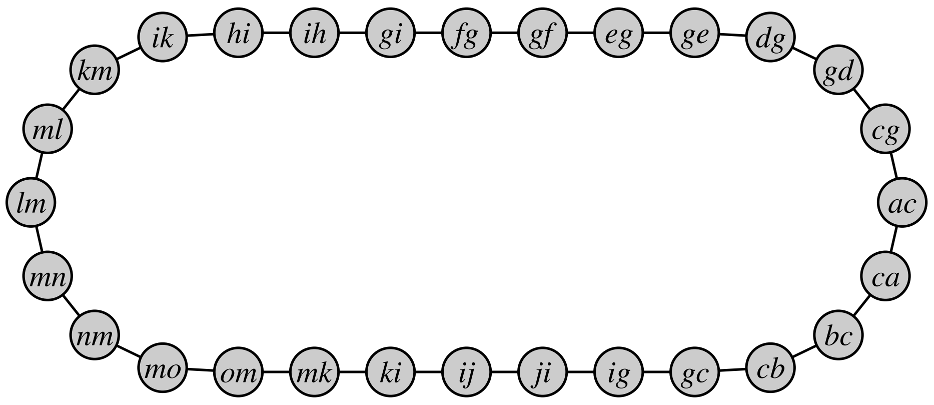


Figure 3.6: The Euler tour (as used by top trees) corresponding to the tree in Figure 2.4. It should be read in counterclockwise order.

The circular order in the original forest induces a unique circular order of the clusters present on the other levels of the contraction. (Recall that the original forest can be regarded as level zero.) Therefore, each level is associated with a unique Euler tour.

Each arc in the Euler tour maintains a pointer to its successor and to its twin arc. With that information, the data structure can easily detect valid moves. Let a be any arc, with b as its successor, a' as its twin, and b' as its successor's twin. Assume that b is different from both a and a'. Arc a can be *raked* onto b if and only if $succ(a') = a$. Looking only at Figure 3.6, we know, for example, that vertex g could be *raked*, since arc fg is immediately followed by gf. Similarly, an arc a can be *compressed* with its successor b if and only if $succ(b') = a'$. In Figure 3.6, we know that vertex k could be *compressed* because $succ(ik) = km$ and $succ(mk) = ki$.

Subtours. At any level of the update algorithm, an *inactive subtour* of an Euler tour $\mathcal{E}$ is a nonempty, maximal, contiguous sublist of $\mathcal{E}$ containing only inactive arcs. A *proper subtour* of an Euler tour $\mathcal{E}$ is an inactive subtour that starts and ends at the same vertex v. In other words, the tail of its *first arc* (the only arc without a predecessor in the subtour) and the head of its *last arc* (the only arc without a successor) are the same vertex v, which we refer to as the *anchor* of the subtour. Note that, if a proper subtour contains an arc (u, w), it will also contain (w, u). A proper subtour represents a subtree of the original tree, and its anchor is the intersection between the proper subtour and its complement (i.e., the sublist of $\mathcal{E}$ containing the arcs that are not in the proper subtour). Figure 3.7 shows the four proper subtours of the second level (level 1) of the contraction shown in Figure 3.5.

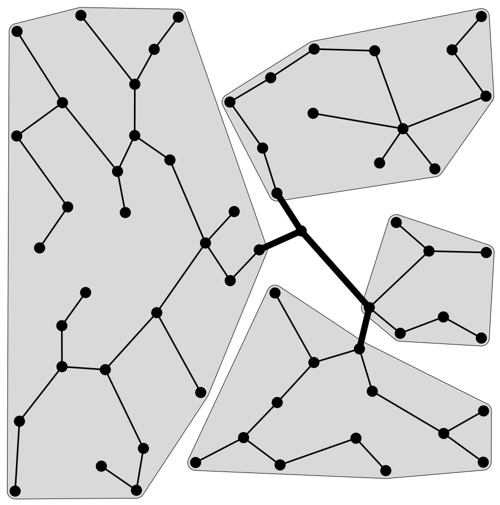


Figure 3.7: The four proper subtours of the second level of the contraction in Figure 3.7.

Intuitively, a proper subtour can be thought of as a generalized leaf of the original tree, since it only touches the remainder of the tree in one vertex. An arbitrary inactive subtour does not necessarily have this property. Take Figure 2.4, and suppose that both (d, g) and (i, k) are active and all other edges inactive. The inactive subtour consisting of arcs km, ml, lm, mn, nm, mo, om, and mk is a proper subtour (anchored at k). In contrast, the inactive subtour consisting of arcs ij, ji, ig, gc, cb, bc, ca, ac, cg is not a proper subtour, since its first vertex (i) is different from the last (g); in particular, ig belongs to the subtour but gi does not.

Our update algorithm is such that the set of active edges within a component is always contiguous, which ensures that every inactive subtour it deals with is also a proper subtour.

3.2.2 Proof Outline

Recall that our goal is to show that the update algorithm performs a constant number of new moves in each level. To that end, we will show that there can be no more than four proper subtours in any level. This means that the core only touches the rest of the tree in four points, which has two consequences: first, the number of new edges added to the core in each level is bounded by a constant; second, the core behaves "almost" like a free tree, in the sense that a constant fraction of its original edges (minus a constant) must be eliminated from one round to the next due to *rakes* and *compresses*. Since the core starts the algorithm with at most one edge, these facts guarantee that it will not grow beyond constant size. To complete the proof, it suffices to show that all new moves performed by the algorithm involve either the core or an edge within constant distance (in the Euler tour) from it.

The remainder of this section makes the main points of this argument more precise.

The notion of proper subtours is helpful because it allows us to identify regions of the tree that need not be processed explicitly by the update algorithm. Intuitively, one expects that a move involving two inactive arcs that are adjacent only to inactive neighbors should have no problem being replicated. In fact, if we state this intuition more formally, it is indeed true:

Lemma 7 *Let $\mathcal{E}_v$ be a proper subtour anchored at v at the beginning of round ℓ. Any new move involving arcs of $\mathcal{E}_v$ at this level must involve the first, the second, or the last arc of this subtour.*

We delay the proof of this lemma until Section 3.2.3. It ensures that the update algorithm only needs to process clusters that are active or are close to the extremes of a proper subtour. All other moves will be replicated, i.e., processed implicitly.

To prove that the number of processed clusters is bounded by a constant, all we need is the following result:

Theorem 1 *During the update algorithm, the inactive edges of the Euler tour at any level can be partitioned into at most four inactive subtours, all of them proper.*

This is clearly true at level zero. When an edge (v, w) is added to (by *link*) or removed from (by *cut*) the forest, it will determine exactly two proper subtours: one anchored at v and the other at w.[3] We shall see, however, that these subtours are *unstable*: either can be split into two proper subtours at a subsequent level. We will show that, whenever an unstable subtour is indeed split, the two resulting subtours will be *stable*: they cannot be further subdivided in two until one of them disappears completely.[4] We say that these two subtours are *coupled*: the existence of one ensures the stability of the other, essentially because the path between the two anchors restricts the types of new moves that can be performed. When one of the subtours disappears (because all of its edges have become active), the other becomes unstable.

More formally, let the *parent set* of a proper subtour be the set of all inactive parents of the clusters represented in the subtour. We shall prove the following facts about stable and unstable subtours:

[3]For simplicity, we consider only these *links* and *cuts* now; *expose* will be analyzed independently later.
[4]A formal definition of stable and unstable subtours will be given in Section 3.2.4.

Lemma 8 *Let $\mathcal{E}$ be an unstable subtour at level ℓ. If the parent set of $\mathcal{E}$ is nonempty, the corresponding arcs at level $\ell + 1$ will either form a single unstable subtour or a pair of coupled stable subtours.*

Lemma 9 *Let $\mathcal{E}_v$ and $\mathcal{E}_w$ be a pair of coupled stable subtours at level ℓ. If both have non-empty parent sets at level $\ell + 1$, the corresponding arcs will form a pair of coupled stable subtours. If only one of the parent sets is non-empty, it will form an unstable subtour.*

Together with the fact that level zero has at most two subtours, both unstable, these lemmas imply that each level of the contraction may have no more than two unstable subtours or four stable subtours (or, combining these two conditions, one unstable and two stable subtours).

This proves Theorem 1, modulo the proofs of Lemmas 7, 8, and 9. These will be given in Sections 3.2.3, 3.2.5, and 3.2.4, respectively. Section 3.2.6 gives more precise bounds on the number of active edges that must be processed by the algorithm in each level.

3.2.3 Replicated Moves

To prove Lemma 7, we need the following result:

Lemma 10 *Let $\mathcal{E}$ be an Euler tour of the new contraction at some level ℓ, and let $\mathcal{E}_v$ be a proper subtour of $\mathcal{E}$ anchored at v. Any original move involving two edges of $\mathcal{E}_v$ will be replicated, with the possible exception of a move that involves both the first and the last arcs of the tour.*

Proof. As already observed, to determine whether a *rake* or *compress* move is valid, we only need to look at the two arcs involved and at their twin arcs. We claim that a valid move involving a and its successor b may cease to be valid only if the successor of a, a' (the twin arc of a), or b' (the twin arc of b) changes. For both *rake* and *compress*, we must still have $succ(a) = b$ in the new contraction. For *rake*, we also need to guarantee that $succ(a') = a$; the successors of b and b' are irrelevant. For *compress*, the move will remain valid only if $succ(b') = a'$; the successors of b and a' are irrelevant.

Suppose both a and b belong to $\mathcal{E}_v$ (which implies that their twins also do). Since $\mathcal{E}_v$ is a proper subtour, the only arc in $\mathcal{E}_v$ that may change its successor is the last one. Any original move that does not involve this arc will therefore be replicated. A move involving the first arc of the tour and its predecessor may not be replicated either, but in this case the predecessor either does not belong to the subtour or coincides with its last edge; therefore, the lemma still holds. □

We are now ready to prove Lemma 7. Assume there is a new move involving two arcs e and f that appear consecutively in $\mathcal{E}_v$ (i.e., f is the successor of e). If either e or f (or any of their twins) is an extreme arc, we are done: the lemma is not violated. So assume that neither of these arcs (or their twins) is the first or the last arc of the subtour. We claim that this situation can only happen when e is the second arc of the tour.

In the original contraction, e and f could not have both stayed unmatched, or else that contraction would not be maximal. Since the move is new, at least one of these arcs must have been involved in some other move, which could not be replicated. Call it the *blocked move*. Because we are assuming that neither e nor f is an extreme arc, both edges involved in the blocked move must belong to $\mathcal{E}_v$. If one of them did not, then the other arc would necessarily be an extreme arc of $\mathcal{E}_v$.

From the proof of Lemma 10, we know that exactly three types of moves may be blocked in this case: (1) *compress*(v), which combines that last arc with the first (this is the case where the successor of a or b' changes); (2) a *rake* of the last arc around v onto the first arc (this is the case where the successor of a changes); or (3) *rake*(v), where the first arc (which is the twin of the last) is *raked* onto its successor (this is the case where the successor of a' changes). The first two cases only involve extreme edges; case (3) is the only one that may involve an arc that is not extreme: the second arc of the subtour. This arc must be e.□

Note that the only possible new move that involves neither the first nor the last arc happens in a very specific situation: when the anchor v was *raked* in the original contraction

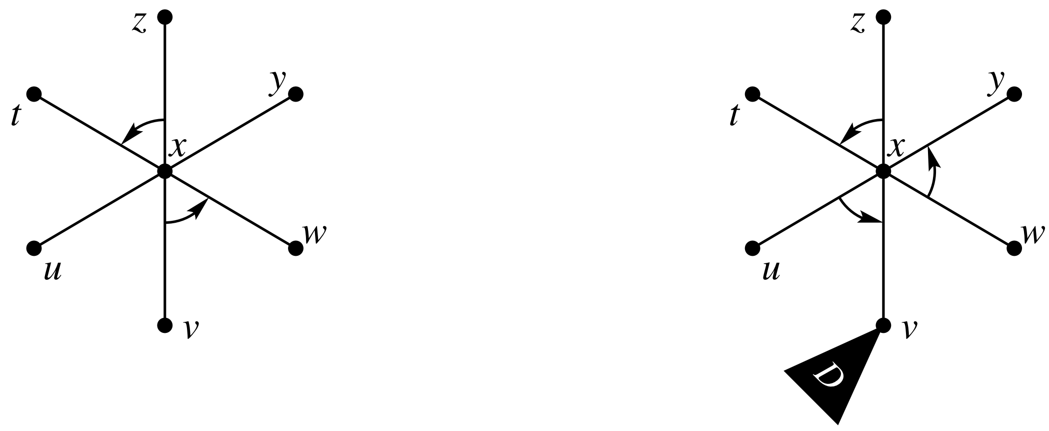


Figure 3.8: The only case in which a new move between inactive edges does not involve either the first or the last arc of a proper subtour. The two configurations refer to the same level, but the one on the left refers to the original contraction and the one the right to the new contraction. The only difference between them is that v has an additional set of active edges incident to it (shown as the subtree D). This makes v the anchor of a proper subtour with first arc (v, x) and last arc (x, v). Because v now has degree greater than one, edge (v, x) can no longer be *raked* onto (x, w). This frees edge (w, x) to be *raked* onto (x, y) as a new move—one that does not involve either the first or the last arc of the tour.

but cannot be in the new one because it has at least one incident active edge. This case is depicted in Figure 3.8. We shall see that, if a subtour is stable, its anchor cannot be *raked* in either contraction. Therefore, the case depicted in Figure 3.8 can only happen when v is the anchor of an unstable subtour.

3.2.4 Stable Subtours

This section is dedicated to the proof of Lemma 9. To that end, we first need a precise definition of stable subtours. Let $\mathcal{E}_v$ be a proper subtour at level ℓ anchored at vertex v. We say that $\mathcal{E}_v$ is *stable* if all of the following conditions are satisfied:

- There exists a vertex $v^* \neq v$ in the same connected component as v in both contractions (old and new) with at least one incident inactive edge.
- In both contractions, the path between v^* and v contains only active edges.
- Let (v, w) be the first edge of $\mathcal{E}_v$: its predecessor around v belongs to the path between

v^* and v. Similarly, the predecessor around v^* of the first arc in the subtour anchored at v^* also belongs to this path.

If there is no vertex v^* satisfying the properties above, $\mathcal{E}_v$ is said to be an *unstable* subtour. Otherwise, $\mathcal{E}_v$ and $\mathcal{E}_{v^*}$ (the proper subtour anchored at v^*) will each be stable, and they will be *coupled*. Figure 3.9 depicts a generic stable subtour.

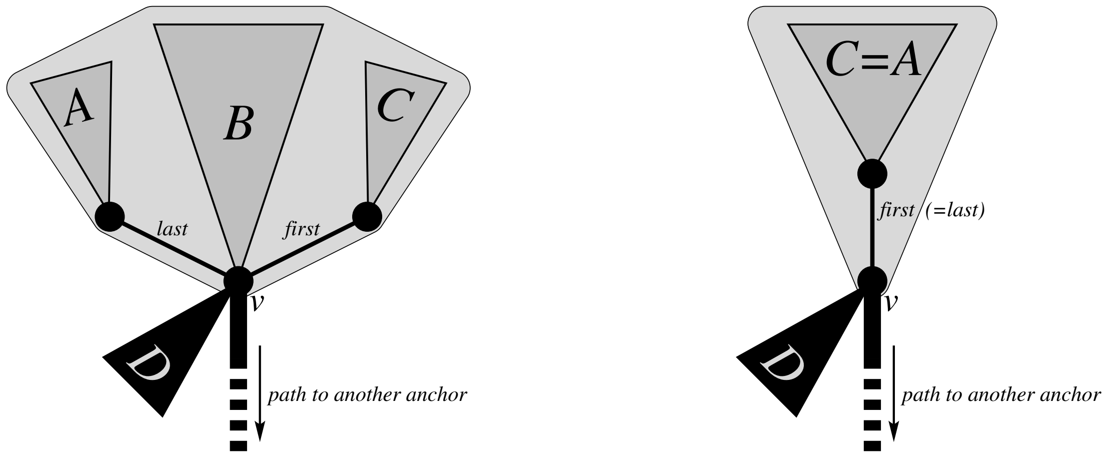


Figure 3.9: The two possible configurations of a generic stable subtour anchored at v. On the left, the first and the last arcs represent two different edges; on the right, they represent the same edge. The subtrees denoted by A, B, and C belong to the stable subtour and contain only inactive edges. The subtree denoted by D does not belong to the stable subtour and contains only active edges. Any of these subtrees (A, B, C, or D) may be absent. There must be at least two edges incident to v: at least one must be inactive and at least one active (the first on the path to another anchor, which consists only of active edges).

We shall prove that a stable subtour cannot be divided into two proper subtours in a single iteration. More precisely, let $\mathcal{E}'$ be the parent set of $\mathcal{E}_v$, i.e., the collection of all clusters on level $\ell + 1$ that are parents of arcs in $\mathcal{E}_v$ and are still inactive. The conditions above ensure that, if $\mathcal{E}'$ is non-empty, it will form a proper subtour.

To prove this, we must establish two facts. We first show that, even though the last arc of the subtour may be involved in a new move, the resulting active parent edge will always belong to the extreme of the inactive subtour (and therefore will not partition it in two). Second, we prove that any new move involving the first arc of the subtour will necessarily involve the last arc as well. This is a direct result of the definition of a proper subtour: the three properties above ensure that the first arc cannot be a receiver around v in either contraction (old or new) and that it cannot be *raked* around its other endpoint (besides v). This limits the possible outcomes for this arc. In particular, it ensures that the case depicted in Figure 3.8 cannot happen, and therefore we only need to worry about new moves involving the first and the last arc.

We will analyze each case in turn in the next two subsections. The analysis consists of enumerating all possible outcomes for each relevant edge of the subtour (first and last), and showing that no outcome violates the lemma. The case analysis is somewhat tedious, but straightforward. We just need to make sure to consider, for each possible outcome for an edge in the original contraction, all possible outcomes in the new contraction. Recall that there are seven possible outcomes for an edge (v, w): *raked* around v, *raked* around w, *compressed* at v, *compressed* at w, receiver around v, receiver around w, and unmatched. Since we must deal with outcomes in both contractions, in principle there are 49 combinations to consider for each edge.

The Last Arc

Let (u, v) be the last arc of a stable subtour anchored at v, and assume that the subtour has at least two edges (if it has only one, it cannot possibly be split in two). We will show that, although there can be a new move involving the last arc of the original subtour, it will never split the subtour in two pieces.

If the original move involving (u, v) is replicated, its parent cluster will be inactive. This will always happen if (u, v) was a receiver around u, *compressed* at u, or a receiver around v (in the latter case, stability ensures that (v, u) is not the first arc). Therefore, we do not need to worry about these cases. In addition, we can discard the case in which (u, v) is *raked* around u: this cannot happen in either contraction, since stability guarantees that v will have degree at least two in both contractions.

In the remaining cases, (u, v) may be involved in a new move, in which case its parent cluster will be active. There are three possible original outcomes for (u, v):

1. (u, v) was originally *raked* around v. This can only happen if u had degree one in the original contraction (and therefore in the new one). This fact eliminates three possible outcomes for (u, v) in the new contraction: (u, v) cannot be *raked* around u, *compressed* at u, or a receiver around u. The only possible outcomes are:

   (a) (u, v) is *raked* around v (onto an active edge). In this case, v continues to be the anchor of the subtour, unless the subtour becomes empty.

   (b) (u, v) is *compressed* at v: since u has degree one, this can only happen if (u, v) is the only edge in the subtour, which will become empty.

   (c) (u, v) either remains unmatched or becomes a receiver around v: the parent edge (also (u, v)) will be active, but, because u has degree one, it will not split the subtour. Vertex v will continue to be the anchor.

2. (u, v) was originally *compressed* at v. Because the subtour is stable, v must have exactly one inactive neighbor in the new contraction, and at least one active neighbor (the first edge on the path to v^*). The parent edge of (u, v) will always be active, and the anchor will move. The following outcomes are possible for edge (u, v) in the new contraction:

   (a) *compressed* at v: u becomes the new anchor of the subtour.

   (b) *compressed* at u: u's other neighbor (besides v) becomes the new anchor of the subtour.

   (c) *raked* around v: this can only happen if (u, v) is the only edge in the subtour, which becomes empty after the move.

   (d) receiver around u or unmatched: u becomes the new anchor of the subtour.

   None of these outcomes splits the original tour. The remaining two outcomes are impossible. Edge (u, v) cannot be *raked* around u because v has degree at least two, and it cannot be a receiver around v because the predecessor of (u, v) around v belongs to the path to v^*, and therefore is not a leaf.

3. (u, v) was originally unmatched. The edge can stay unmatched in the new contraction as well, in which case its parent will be inactive. It cannot be *compressed* at u or a receiver around u, since these moves could have been performed in the original contraction. The edge cannot be a receiver around v either, since stability ensures that the edge *raked* onto it must be inactive, and therefore the move could have been performed in the original contraction. Also, it cannot be *raked* around u, since stability guarantees that v has degree greater than one. The only possible new moves are:

   (a) (u, v) is *raked* around v: v remains the anchor of the subtour.

   (b) (u, v) is *compressed* at v: u becomes the new anchor of the subtour.

   The subtour will disappear if the new anchor has no remaining inactive neighbors.

The First Arc

It remains to be shown that moves involving the first arc will not split the tour either. Let (v, w) be the first arc of a stable subtour anchored at v.

If the original subtour has only one edge, either (v, w) will remain unmatched in both contractions (in which case the whole tour will survive) or its parent edge will be active (in which case the subtour will simply disappear). The analysis that follows handles the nontrivial case, in which the original subtour has at least two edges.

Of all possible original outcomes for (v, w), three will always be replicated: *raked* around v, receiver around w, or *compressed* at w. Stability ensures that two original moves are impossible: (v, w) cannot have been a receiver around v (since its predecessor belongs to the path to another anchor), and it cannot be *raked* around w (since v has degree greater than one in both contractions). This leaves only two original outcomes to be considered: (v, w) being *compressed* at v or remaining unmatched. We consider each case in turn.

First, assume (v, w) was originally *compressed* at v. If this is the case, v's original degree was two. Its degree in the new contraction must be at least two, with a single inactive edge incident to it, (v, w) itself. The edge corresponding to the second original inactive edge must be on the path to the other anchor. This means that edge (v, w) corresponds to both the first arc and the last arc of the subtour, and we have already seen that moves involving the last arc never split the subtour.

Now consider what may happen in the new contraction if (v, w) was originally unmatched:

1. (v, w) remains unmatched in the new contraction: the parent will be inactive.
2. (v, w) is *compressed* at v: this means that v has degree two, and therefore (w, v) is the last arc of the subtour; as already observed, this new move will not split the subtour.
3. (v, w) is *raked* around v: this can only happen if the target is the last edge of the tour; otherwise, the move could have been performed in the original contraction.

All other outcomes for (v, w) are impossible. It cannot be a receiver around w or *compressed* at w, since those moves could have been performed in the original contraction (and therefore the edge should not have remained unmatched). It cannot be *raked* around w because v has degree greater than one. Finally, it cannot be a receiver around v because its predecessor around v belongs to the path from v^* to v (and therefore is not a leaf).

Completing the Proof

The case analysis above has shown that a stable subtour at level ℓ cannot be split into two parts in a single round: its parent set will either be empty or form a proper subtour at level $\ell + 1$. To finish the proof of Lemma 9, all we have to do is prove that, in the latter case, the proper subtour at level $\ell + 1$ will also be stable.

Let v_ℓ and v_ℓ^* be the anchors of a pair of coupled subtours at level ℓ, and let $v_{\ell+1}$ and $v_{\ell+1}^*$ be the anchors of the corresponding subtours at level $\ell + 1$ (we may have $v_\ell = v_{\ell+1}$ and $v_\ell^* = v_{\ell+1}^*$, but not necessarily). Because the original tours are stable, v_ℓ and v_ℓ^* belong to the same connected component in the original contraction, and the path between them contains active edges only. Since moves cannot disconnect the graph, the same will be true for $v_{\ell+1}$ and $v^*_{\ell+1}$ at level $\ell + 1$.

It remains to prove that the predecessor of the first arc of each subtour does belong to the path between $v_{\ell+1}$ and $v^*_{\ell+1}$ (in both contractions). This follows from the case analysis: whenever there is a new move involving the first edge of a stable subtour at level ℓ, it must involve its last edge as well. This will cause the anchor to change. Only one edge incident to the new anchor will participate in a move at level ℓ, and its parent will belong to the path to the other anchor at level $\ell + 1$. No other edge around the new anchor will be active, which implies that the first arc of the proper subtour must have the path to the other anchor as its predecessor in both contractions.

This completes the proof of Lemma 9. □

3.2.5 Unstable Subtours

We now prove Lemma 8. We will show that (1) only a move involving the first arc of an unstable subtour may cause it to be split; and (2) if such a split does occur, it will always create a pair of stable subtours. Once again, the proof is based on case analysis. For reference, Figure 3.10 presents a generic representation of an unstable subtour.

The analysis will consider two types of rounds: *standard rounds*, in which the anchor v does not disappear in either contraction (old and new), and *special rounds*, in which v disappears in at least one of the contractions. During a standard round, only the first arc can split the tour; a move involving the last arc will split the tour only if it involves the last arc as well. Since the anchor is not allowed to disappear, we do not need to worry specifically about a new move involving the second arc during a standard round—if it happens, the first arc will be involved as well. The second arc will be considered during the analysis of special rounds, however.

In several of the cases analyzed in this section, the original subtour will be split into

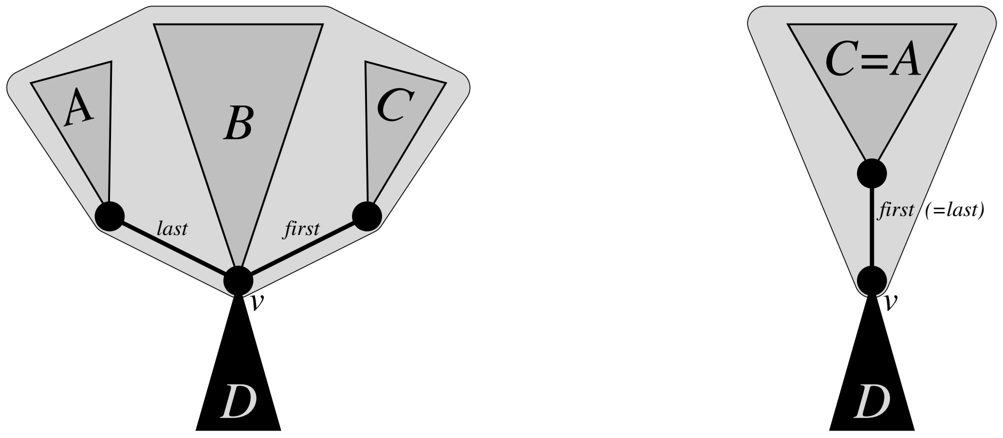


Figure 3.10: The two possible configurations of a generic unstable subtour anchored at v. On the left, the first arc and the last arc represent different edges; on the right, they represent the same edge. The subtrees denoted by A, B, and C belong to the unstable subtour and contain only inactive edges. The subtree denoted by D does not belong to the unstable subtour and has only active edges incident to v. Any of these subtrees (A, B, C, or D) may be absent. In particular, there may be no active edge incident to the anchor v.

two subtours. These subtours will always be stable and coupled, but we will not prove this in every case. The reader can easily verify in each case that (1) the anchors of the two subtours are connected by a path containing only active edges; and (2) the last active edge in the circular order around each anchor belongs to this path. This is enough to ensure the subtours are stable. Of course, if one of the subtours created is empty, the other will actually be unstable. If both are empty, the original subtour simply disappears.

Standard Rounds: First Arc

We first consider what happens with the first arc (v, w) of an unstable subtour anchored at v during a round in which v does not disappear in either contraction.

If the original move involving (v, w) is replicated, the parent edge will be inactive, and therefore the subtour cannot be split as a result. We only need to worry about the cases where the parent of (v, w) is active. During a standard round, the following outcomes for (v, w) in the new contraction may result in an active parent: (1) unmatched; (2) *compressed* at w; (3) *raked* around v; (4) receiver around v; and (5) receiver around w. Any of these five outcomes is possible, for instance, if the arc was a receiver around v in the original contraction.

In all these cases, let (v, x) be the new parent edge of (v, w). It is active. There are three cases to consider:

1. If both x and v have at least one inactive neighbor at the end of the round, the original tour will be split in two, and x and v will each be the anchor of a stable subtour.

2. If v ends up with at least one inactive neighbor and x with none, the original subtour will not be split and v will remain the anchor of an unstable subtour. The case in which x ends up with at least one inactive neighbor and v does not is symmetric, the only difference being that x becomes the new anchor.

3. If both x and v end up with no inactive neighbors, the subtour simply disappears.

Standard Rounds: Last Arc

We now consider the last arc (u, v) of a stable subtour rooted at v during a standard round. It can split the tour when involved in a move with the first arc. We have seen, however, that if a move involving the first arc splits the subtour, the result will be a pair of coupled stable subtours. Therefore, we only need to worry here about the cases where (u, v) is involved in a new move by itself. In such cases, we shall see that tour will not be split. The possible original outcomes for (u, v) during a standard round are:

1. (u, v) is *compressed* at u or a receiver around u: the move will be replicated.

2. (u, v) is *raked* around v: This means u has degree one. The only possible outcomes in the new contraction during a standard round are:

   (a) (u, v) is *raked* around v: its target is either already active (in which case the subtour will not be split by the new move) or the first arc of the subtour (which we have already considered).

   (b) (u, v) is a receiver around v or remains unmatched: the parent edge becomes active, but the original tour will not be split, because u has degree one.

   The other cases either are impossible: they either cause v to disappear (which cannot happen during a standard round) or require u to have degree greater than one.

3. (u, v) is a receiver around v: the move will be replicated, unless the predecessor of (u, v) around v is active. But this can only happen if (u, v) is the only inactive edge incident to v, which means that it is also the first arc of the stable subtour. This case has already been considered.

4. (u, v) remains unmatched: In the new contraction, the edge cannot be a receiver around u or *compressed* at u, since these moves could have been performed in the original contraction. The only possible outcomes during a standard round are:

   (a) (u, v) is a receiver around v: This can only happen if either (u, v) or its predecessor are the first arc of the subtour, otherwise the move could have been performed in the original contraction. In either case, the new move will involve the first arc.

   (b) (u, v) is a *raked* around v: If the target is an active edge, the subtour will not be split; if the target is inactive, it must be the first arc of the subtour.

   (c) (u, v) remains unmatched: the edge remains inactive.

   The remaining two cases cause v to disappear, which cannot happen during a standard round.

The case analysis above has shown that the last edge of a subtour will never split the tour, unless it is also the first edge or participates in a move with the first edge.

Special Rounds

We now consider *special rounds*, in which the anchor v does disappear in at least one of the contractions. The analysis uses the fact that, when v has no incident active edge in one of the contractions (old or new), it must have at least one additional incident edge in the other contraction. If this were not true, both contractions would be exactly the same.

The possibilities for v are as follows:

1. v is *raked* in the new contraction (regardless of what happens in the original contraction). Let (v, w) be the only edge incident to it in the new contraction at the beginning of the round, and let (w, x) be the edge onto which (v, w) is *raked.* This must be a new move, since v has degree at least two in the original contraction, and (w, x) must have been originally unmatched. Both w and x will become anchors of coupled stable subtours.

2. v is *compressed* in the new contraction: Let (v, w) and (v, u) be the edges adjacent to v in the new contraction. Without loss of generality, assume (v, u) is inactive—at least one of the edges must be, or else v would not be an anchor. There are two cases to consider:

   (a) (v, w) is also inactive: The move must be new, since v has degree greater than two in the original contraction. In addition, (v, u) must not have been involved in a move around u and (v, w) must not have been involved in a move around w, since such moves would have been replicated. After the move, both u and w become anchors of coupled stable subtours.

   (b) (v, w) is active: The subtour will not be split; it will remain stable with u as its new anchor.

3. v does not disappear in the new contraction but is *raked* around its neighbor x in the original contraction. This is the case depicted in Figure 3.8. In the new contraction, (x, v) is the only inactive edge incident to v, and there must be at least one active edge incident to it as well. Let (x, w) be the receiver of the original *rake*; it is inactive in both contractions. The parents of (x, v) and (x, w) in the new contraction will be active.

If (x, w) is *compressed* at x in the new contraction, then w becomes the anchor of an unstable subtour. In this case, (x, v) and (x, w) will have the same parent.

If x is not *compressed*, (x, v) will either remain unmatched or become a receiver; in either case, the parent edge will be active. (In Figure 3.8, it becomes a receiver around x.) Edge (x, w) can stay unmatched, be *compressed* at w, become a receiver around w, or be *raked* around x (the latter is what happens in Figure 3.8). In any of these cases, let y be the other endpoint (besides x) of the parent cluster of (x, w). At the end of the round, both x and y will be anchors of coupled stable subtours, unless one of the subtours is empty, in which case the other vertex will be the anchor of an unstable subtour.

4. v does not disappear in the new contraction, but is *compressed* in the original contraction. There are two cases to consider, depending on the number of inactive edges incident to v in the new contraction at the beginning of the round:

   (a) One inactive edge (v, w). Regardless of the outcome of this edge, its parent will be active in the new contraction. Since we are assuming v does not disappear, the possible outcomes for (v, w) are:

      i. receiver around w, receiver around v, or unmatched: w becomes the anchor of the unstable subtour;

      ii. *raked* around v: this can only happen when w has degree one, and the subtour simply disappears;

      iii. *compressed* at w: w's other neighbor (besides v) becomes the anchor of the unstable subtour.

   (b) Two inactive edges, (u, v) and (v, w). Without loss of generality, suppose (u, v) immediately precedes (v, w) in the tour. The successor of (v, w) is active, since there must be at least one additional edge incident to v in the new contraction.

   If (u, v) is *raked* onto (v, w), w will become the anchor of an unstable subtour. If (v, w) is itself *raked*, the receiver will be active and the subtour will not be split.

Its anchor will be u, unless (u, v) is combined (by *compress*) with some other edge (u, x). In this case, the new anchor will be x.

If neither (u, v) nor (v, w) is *raked* around v, each will participate in a move by itself, and two active parents will be created. More precisely, (v, w) may be *compressed* at w, a receiver around w, or unmatched. Similar outcomes are possible for (u, v), which can also be a receiver (of an active edge) around v. In every case, the endpoint of each parent cluster that is different from v will become the anchor of a stable subtour, and the two subtours will be coupled.

Completing the Proof

In the analysis of standard rounds, we have shown that only a move involving the first edge of an unstable subtour may cause it to be split in two, but, whenever this happens, the subtours thus created will be coupled and stable. For special rounds, we have shown that an unstable subtour may be split into at most two parts and, when there are two, the resulting subtours will also be coupled and stable. This proves Lemma 8. □

3.2.6 Running Time

Now that we have proved the lemmas necessary to establish Theorem 1, we can finally bound the number of clusters that need to be processed by the update algorithm. Recall (from Section 3.2.1) that the *core* at level ℓ is the subgraph induced by all active edges in the new contraction at that level, and that the *core image* is the subgraph induced by the active edges in the original contraction.

Lemma 11 *At each level, the sizes of the core and of the core image are bounded by a constant.*

Proof. When considered as an isolated graph, the core is a forest (and so is the core image). Lemma 4 would guarantee that a fraction of at most q of its original vertices will remain after each round, with $q = 5/6$. The lemma cannot be applied in full, however: some of the edges that look like candidates for *rake* and *compress* may not actually be eliminated, either because their degree is higher in the actual tree or because their actual successors or predecessors do not belong to the core. What the lemma does guarantee is that, if the number of such "false candidates" is at most some constant k_b, then at least $[(1-q)s] - k_b$ vertices will be eliminated after a round (s being the original number of vertices). We must also take into account the fact that more edges may be incorporated into the core from one round to the next. We will show that there is a constant upper bound k_a on the number of such additional edges.

If s_i is the number of edges in the core after round i, the following recurrence relation holds for all $i > 0$:

$$s_i \quad \leq \quad qs_{i-i} + k_a + k_b.$$

Its solution is

$$s_i \leq q^i s_0 + \frac{(k_a + k_b)(1 - q^i)}{1 - q},$$

where s_0 is the number of edges in the core before the update procedure begins. Its value is one for *links* and zero for *cuts*. Using the fact that $q = 5/6$, s_i can be upper bounded by

$$s_i \leq s_0 + \frac{k_a + k_b}{1 - q} = 1 + 6(k_a + k_b), \tag{3.2}$$

which is a constant. □

We have shown that, in any round, there are no more than four proper subtours touching the core (or the core image). More precisely, the following combinations are possible: (a) two pairs of coupled stable subtours; (b) one unstable subtour and one pair of coupled stable subtours; (c) two unstable subtours; (d) one pair of coupled stable subtours; (e) one unstable subtour; and (f) no subtour (all edges are active). For the purpose of bounding k_a and k_b, the first three cases dominate the other three.

Bounding k_a. A *core expansion* at level ℓ occurs when an active cluster with only inactive children (i.e., inactive clusters at level $\ell - 1$) is created. Similarly, a *core image expansion* at level ℓ occurs whenever a cluster of the core image at level ℓ has only inactive children. An *expansion cluster* at level ℓ therefore represents a move that happens in only one contraction and does not involve any active edges. These are the clusters we have to count to bound k_a. All other clusters are irrelevant: an inactive cluster will not belong to the core, and we count an active cluster with an active child as belonging to the core already—although technically the parent belongs to the core at one level, and the child to the core at the previous level.

Consider the core first. At most one expansion cluster will be created per stable subtour, since any new move must involve its last edge. For each unstable subtour, there can be no more than two expansion clusters. In general, one expansion cluster will be the parent of the first edge and another the parent of the last. The configuration depicted in Figure 3.8 is a special case: one expansion cluster is the parent of the second edge, and another the parent of both the first and last edges, which coincide. Since there can be at most two unstable subtours or at most four stable subtours, the core may have at most four expansion clusters.

Now consider the core image. We must count the number of clusters that have only inactive children and must be deleted.

On a stable subtour, there can be at most one of those: the parent of a previously unmatched edge that becomes matched as a new move involving the last arc. No *rake* or *compress* involving only edges of a stable subtour will be blocked, and therefore the cluster representing such a move cannot be an expansion cluster.

On an unstable subtour, however, an original move involving only inactive edges can be blocked (and the cluster representing it will be an expansion cluster). Furthermore, each of the clusters originally involved in this move can be combined with a previously unmatched edge. The original dummy parents of these edges will also be expansion clusters. As a result, there can be up to three expansion clusters in the core image.

This happens, for instance, in the situation depicted in Figure 3.8. An edge (x, v) is originally *raked* around x but the move cannot be replicated because the anchor v has one additional incident edge in the new contraction. Let (x, w) be (x, v)'s successor around x (the original target edge), and let (x, u) be (x, v)'s predecessor. As shown in Figure 3.8, both (x, u) and (x, w) may be *raked* as new moves. In this case, the original dummy parents of (x, u) and of (x, w)'s successor (which is (x, y) in the picture) will be expansion clusters of the core image, as will the original parent of (x, v) and (x, w).

In the worst case, when there are two unstable subtours, the core image will have six expansion clusters, which means that $k_a \leq 6$.

Bounding k_b. Recall that k_b is an upper bound on the number of moves that are assumed to be possible by Lemma 4 but end up being blocked because the core and the core image are not isolated trees.

First, consider the core. A stable subtour may block only one move, involving the successor of its last arc. Each unstable subtour, on the other hand, may block no more than two moves: one involving the successor of the subtour, and another involving the predecessor. The same analysis holds for the core image. Therefore, we can ensure that $k_b \leq 4$.

Final bound. Recall from the proof of Lemma 11 that $s_i \leq 1 + 6(k_a + k_b)$. For the core, $k_a + k_b = 8$, which means that it will contain no more than 49 clusters in any given level. For the core image, $k_a + k_b = 10$, which bounds its number of clusters by 61.

Theorem 2 *A contraction can be updated in $O(\log n)$ worst-case time after a* link *or* cut*.*

Proof. To update a level, we only have to deal with clusters in the core or the core image, inactive clusters that are adjacent to them, and sometimes with the immediate successor of one of these inactive clusters in the Euler tour. Lemma 11 ensures that both the core and the core image have a constant number of clusters. Moreover, the inactive edges of the graph will be partitioned into at most four proper subtours. Since only the first two arcs and the last arc of each subtour must be checked for new moves, we have to deal with a constant number of clusters per level. The number of levels in the top tree is logarithmic (by Lemma 1), so the total running time of the update procedure is $O(\log n)$. □

The same bound holds for *expose*, as Section 3.3.5 will show.

3.3 Implementation

The update algorithm has been described as building a new contraction that resembles the original contraction as much as possible. In practice, of course, we need to modify the top tree representing the original contraction to make it represent the new one, and we must do so in $O(\log n)$ time. This section describes how this can be achieved.

3.3.1 Representation

The main structure we maintain is the top tree itself. Each cluster is represented as a separate record, with pointers to its parent and its children (of which there can be zero, one, or two). The cluster also stores the user-defined, application-specific data.

To implement the updating algorithm, we also maintain an Euler tour of each level of the top tree. This is a double-linked list of arcs. Each cluster in the level is associated with two arcs, one in each direction. Since each cluster must have access to these two arcs, we actually make the arcs part of the record that represents the cluster. Each arc a has three pointers: to its successor in the Euler tour, to its predecessor in the tour, and to the cluster itself.[5] We denote these fields by $\mathit{succ}(a)$, $\mathit{pred}(a)$, and $\mathit{cluster}(a)$, respectively. Each arc a also needs access to its twin arc (which we denote by $\mathit{twin}(a)$), but an explicit pointer is not needed for that: the twin can be retrieved from the cluster. We also store with each arc the identifier of its head. These fields are also used by the cluster to determine its endpoints.

[5]In a sufficiently low-level programming language, one could actually do without a pointer to the cluster; since the arc is represented inside the cluster, the cluster could be retrieved with some pointer arithmetic.

3.3.2 Identifying Valid Moves

With this representation, identifying valid moves is straightforward. Consider an arc a and its successor b on the tour. To test whether a and b define a valid move, we must first verify that $\mathit{cluster}(a) \neq \mathit{cluster}(b)$ and that both clusters are *free*, i.e., they do not participate in a move. More concretely, a cluster is free if its parent cluster is *null*, dummy, or deleted.

If both conditions are satisfied, we perform the tests outlined in Section 3.2.1. We know that $\mathit{cluster}(a)$ can be *raked* onto $\mathit{cluster}(b)$ if and only if $\mathit{pred}(a) = \mathit{twin}(a)$. Similarly, $\mathit{cluster}(a)$ and $\mathit{cluster}(b)$ can be *compressed* if and only if $\mathit{succ}(\mathit{twin}(b)) = \mathit{twin}(a)$.

3.3.3 Updating the Tree

We update the tree in a bottom-up fashion, starting from the base level. To update a level ℓ, all we need is a list $\mathcal{I}$ of clusters to be inserted and a list of $\mathcal{D}$ of clusters to be deleted. Processing each level requires four sequential steps:

1. Remove from the tree (and the Euler tour) all clusters in $\mathcal{D}$.

2. Insert into the tree (and the Euler tour) all clusters in $\mathcal{I}$.

3. Verify if any previously valid move becomes invalid.

4. Perform new moves until the contraction is maximal.

While we execute the steps above, we must also fill two lists, $\mathcal{D}'$ and $\mathcal{I}'$. Initially empty, they will contain the clusters to be deleted from and inserted into level $\ell + 1$. After all steps above are completed, we set $\mathcal{I} \leftarrow \mathcal{I}'$ and $\mathcal{D} \leftarrow \mathcal{D}'$ and start processing the level above (unless both $\mathcal{I}'$ and $\mathcal{D}'$ are empty, in which case update of the entire tree will be complete).

To perform the last two steps of the algorithm efficiently, we maintain yet another list on each level, a *neighbor list* (denoted by $\mathcal{N}$). This list starts each level empty, but will eventually contain all original clusters that are inactive (i.e., not themselves inserted or deleted) but are close enough to the core or core image to merit special attention. More precisely, an inactive cluster will be inserted into the list if its successor or predecessor (in either the new or the old contraction) is active, or if the original move it was involved in is no longer valid. This second criterion addresses the special case depicted in Figure 3.8.

The following subsections analyze each step in turn.

Step 1: Removal

The first step of the algorithm at level ℓ is to remove every cluster C in $\mathcal{D}$ from the top tree and from the Euler tour. If the twin arcs representing the cluster in the Euler tour are a and b, we remove them from the tour by directly connecting $\mathit{pred}(a)$ to $\mathit{succ}(b)$ and $\mathit{pred}(b)$ to $\mathit{succ}(a)$. We also insert the original parent of C into $\mathcal{D}'$: since C is being deleted, its ancestors must also be. The clusters adjacent to C are added to $\mathcal{N}$, unless they belong to $\mathcal{D}$ already.

Step 2: Insertion

After the first step, the Euler tour at level ℓ will contain only inactive clusters. The second step inserts the new clusters (those in $\mathcal{I}$) into the Euler tour. The nontrivial aspect of this procedure is to determine the appropriate insertion positions.

When updating the base level, the user can specify explicitly where in the circular order a new edge (v, w) is to be inserted. For this, it suffices to specify two predecessor arcs, one having v as its head and the other having w as its head.

To process level ℓ (with $\ell > 1$), we must use level $\ell - 1$ (which will already have a complete Euler tour) to determine the appropriate position of each new arc. Let (v, w) be the arc to be inserted into the Euler tour, and let C be the cluster associated with it. To determine the successor (or predecessor) of (v, w), we must look at the successor (or predecessor) of one of C's children, then take the appropriate arc of its parent.

More precisely, to determine the successor of arc (v, w) at level ℓ, we must look at the appropriate child cluster A of C. If C is a dummy cluster, A is C's only child; if C is a *rake* cluster, A is C's right child (representing the target cluster); if C is a *compress* cluster, A is the only child of C that has w as an endpoint. Let a be the arc of A that has w as its head.

Let b be a's successor, let B be the cluster associated with b, and let P be B's parent. Note that B belongs to level $\ell - 1$ and P to level ℓ. The successor of (v, w) is the arc of P that has w as its tail.

The procedure to determine the predecessor of (v, w) at level ℓ is similar. We must also look at an appropriate child A of C. If C is a dummy cluster, A is its only child; if C is a *rake* cluster, C is its right child; if C is a *compress* cluster, A is the only child that has v as an endpoint. Let a be the arc of A that has v as its tail. Let b be a's predecessor, let B the cluster associated with b, and let P be B's parent. The predecessor of (v, w) is the arc of P that has v as its head.

The strategy outlined above is generic enough to allow arcs on the same level to be inserted in any order. In particular, an arc may be linked to its predecessor and its successor even if these have not been inserted into the tour yet.

Identifying where each new arc should be inserted takes constant time, but the constants involved can be quite high. To avoid doing more work than necessary, at the beginning of the step we set the predecessor and the successor of each arc that needs to be inserted to *null*. When we are about to find the successor of an arc a, we first check if its successor is still *null*. If not, there is no need to apply the procedure above, since the successor will have already been determined (when the successor itself was inserted into the tour and determined that its predecessor was a). Similarly, we only need to actively compute the predecessor of a if its current value is *null*.

During the insertion procedure, we must insert the clusters corresponding to the predecessor and to the successor of each new arc into $\mathcal{N}$, unless they belong to $\mathcal{I}$ already.

Step 3: Detecting Invalid Moves

After the first two steps of the algorithm, the updated Euler tour at level ℓ will be complete. The third step verifies if the original moves involving only inactive edges are still valid. For a move to become invalid, at least one of the vertices involved in the move must have had its neighborhood changed. This means it suffices to check whether the clusters that currently belong to $\mathcal{N}$ (inserted in the previous two steps) still participate in valid moves. Each cluster is tested explicitly using the procedure outlined in Section 3.3.2. If a move is deemed invalid, it may happen that one of the clusters involved in the move is not in $\mathcal{N}$ yet (if it does not touch the core or core image). In this case, we add it to $\mathcal{N}$. Also, the parent cluster representing an invalid move must be inserted into $\mathcal{D}'$.

Step 4: New Moves

The fourth step of the algorithm performs a maximal set of new moves. It does so by explicitly testing (in any order) each cluster in $\mathcal{I}$ and every cluster in $\mathcal{N}$ that is not already matched. For every new move, a new parent cluster P is created, the pointers between P and its children are initialized, and P is inserted into $\mathcal{I}'$. If a cluster involved in a new move had a dummy parent, the parent is inserted into $\mathcal{D}'$. After all clusters are tested, for every cluster in $\mathcal{I}$ and $\mathcal{N}$ that remains unmatched we create a new dummy parent cluster P, initialize the pointers between P and the unmatched cluster, and insert P into $\mathcal{I}'$.

3.3.4 Other Details

List management. As described, the algorithm may try to insert a cluster into more than one list, or into the same list more than once. To avoid multiple insertions, we need to know whether the cluster already belongs to the list of not. Traversing the list before every insertion would be too expensive (although it would still take constant time). A better approach is to make each cluster remember whether it belongs to a list or not. Note that a single bit per cluster suffices for this, since during a call to the update algorithm every cluster can belong to at most one list. This is obvious for new clusters: they can only belong to the insertion list of their level. Existing clusters, on the other hand, can belong either to a deletion list or to a neighboring list. Recall that, if a cluster on level ℓ is ever inserted into a deletion list, this will happen during the update of level $\ell - 1$; when we finally get to level ℓ, the cluster will not be inserted into the neighbor list if it is already marked, regardless of whether the mark is due to a deletion, to an insertion, or to the fact that the cluster belongs to $\mathcal{N}$ already.

Managing values. As the update algorithm progresses, we must call the internal top tree functions in the appropriate order. Whenever the algorithm decides that a cluster must be deleted, we must actually mark all of its ancestors as "scheduled for deletion" and call *split* on each of them in a top-down fashion. This ensures that, whenever *split* is called on a cluster, it will be the root of its tree. Note, however, that the clusters will not be actually deleted (or removed from the Euler tour) until the level to which they belong is processed. When a new cluster is created, on the other hand, we can immediately call *join* to initialize its value, since it will be a root at this point.

Handles. Each vertex in the tree maintains a pointer to one of its incoming arcs on the base level. We call this arc the *handle* of the vertex, and it is *null* if the vertex has degree zero. The handle allows us to go from a vertex to a cluster in the tree that is incident to that vertex.

The handle is also useful when inserting a new arc into the base level. A new arc (v, w) is inserted right after $handle(w)$ on the Euler tour. To set the exact position in which a new arc must be inserted around w, the user can explicitly define which of the existing incoming arcs should be w's handle.

One must be careful to keep the handle updated when an edge (v, w) is removed from the tree by the *cut* operation. If (v, w) happens to be handle of w, we set $handle(w) \leftarrow pred(twin(v, w))$. If this is also (v, w) (which can happen only when w had degree one before the *cut*), we set $handle(w) \leftarrow null$.

3.3.5 Implementing Expose

Exposing a vertex is to ensure it does not disappear in the contraction. When building a contraction from scratch, it is easy to take exposed vertices into account. One just marks them as being exposed and modifies the routines that test for valid moves. Besides the conditions listed in Section 3.3.2, a valid move must be such that the disappearing vertex is not exposed.

In our case, we must assume we already have a valid contraction when we decide to expose a vertex v. We consider two different ways of exposing a vertex: rebuilding the original contraction or building a temporary tree.

Rebuilding the Original Contraction

The obvious way to implement *expose* is to rebuild the original contraction so that the exposed vertices are never eliminated. A first step to achieve this is to change the routine that tests whether a move is valid, as mentioned above.

To expose a vertex v, we must first find the cluster P that represents the move that makes v disappear in the original contraction. (If there is no such cluster, there is nothing to be done: v is already exposed, and we just mark it as such.) To find P, we start with a base cluster that has v as an endpoint (we can use *handle*(v) to find such a cluster) and follow parent pointers until we reach the first (i.e., lowest) cluster that does not have v as an endpoint. Let ℓ be the level that contains this cluster, and let the children of P be A and B. We must mark v as exposed and call the update procedure from level $\ell - 1$ with $\mathcal{I} = \emptyset$, $\mathcal{D} = \emptyset$, and $\mathcal{N} = \{A, B\}$. With these parameters, no cluster will actually be inserted into or removed from level $\ell - 1$. New moves involving clusters at this level will be performed, however, and as a result deletions and insertions will occur at level ℓ and above.

Once v is marked as exposed, future changes to the tree (for instance, when we want to expose a second vertex) will never make a move that eliminates v.

Of course, vertex v cannot remain exposed forever. We might want to expose another vertex u instead of v in the same component. Before that, we must *unexpose* v. The implementation of this operation is very similar to that for *expose*. Starting from a base cluster that has v as endpoint, we follow parent pointers until we reach a level ℓ in which a cluster C containing v could have been involved in a valid move but remained unmatched (because v was exposed). We then mark v as not exposed and start the update procedure from level ℓ with $\mathcal{I} = \emptyset$, $\mathcal{D} = \emptyset$, and $\mathcal{N} = \{C\}$.

Updating the contraction from a level $\ell > 0$ during *expose* or *unexpose* takes no more time than updating the contraction from the base level after a *link*. One just has to consider the clusters in $\mathcal{N}$ to be the core of that of level ℓ. In both *expose* and *unexpose*, the clusters that are not in $\mathcal{N}$ define at most two proper subtours in the component. From this point on, the update algorithm proceeds as before.

Moreover, because the number of exposed vertices in a component is never greater than two, one can still guarantee that, from a level with n nodes, at least $n/6 - 2$ vertices will be eliminated. This means that the height of the top tree is still logarithmic.

Building a Temporary Tree

In [10], Alstrup et al. suggest a much simpler implementation of *expose*. They observe that it can be implemented independently of *link* and *cut*, as long as the height of the top tree is guaranteed to be logarithmic (as is the case here). They use the fact that, even though a vertex v can appear as an endpoint of up to $\Theta(n)$ clusters, it can only appear as an internal vertex in at most one cluster per top tree level.

This suggests the following implementation of *expose*(v, w). First, we call *split* on all clusters that have v or w as internal vertices; there will be at most $O(\log n)$ of those. The result will be a collection of $O(\log n)$ root clusters that partition the edges of the original tree among themselves. Interpreting each root cluster as an individual edge (even though it may actually represent a path in the original tree), we can then build a contraction from scratch in such a way that v and w are exposed, which takes $O(\log n)$ time.

More concretely, Figure 3.11 shows the same top tree as Figure 2.6, with the clusters that would be split during a call to *expose*(c, k) shown in white. The roots of the remaining subtrees can be thought of as representing edges (c, g), (b, c), (e, g), (g, i), (h, i), (m, o), (i, k), (k, m), and (m, n). The *expose* procedure would build a top tree representing a contraction of the free tree induced by these $O(\log n)$ edges only.

Note that we do not have to worry about keeping the entire tree balanced at this point. There are only $O(\log n)$ root clusters and each is guaranteed to have $O(\log n)$ height, which

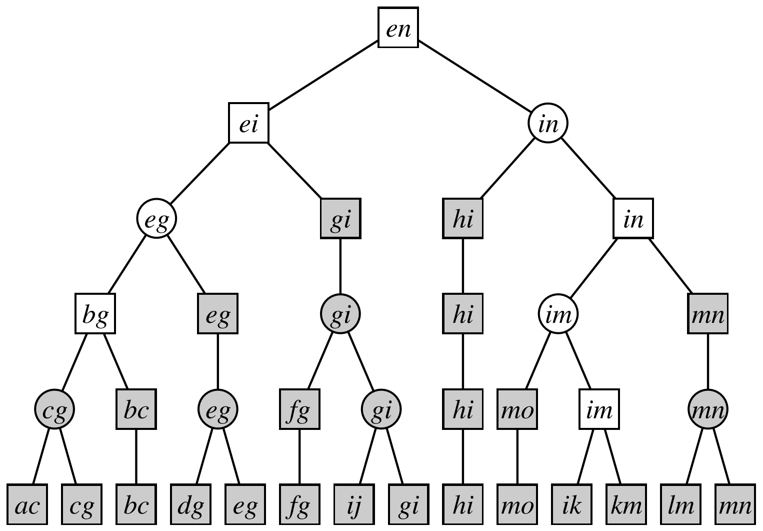


Figure 3.11: Top tree corresponding to the contraction in Figure 2.6, with clusters containing c and k as internal vertices shown in white. These would be split during a call to $expose(c, k)$.

means that the height of the new top tree will be $O(\log n)$ regardless of the order in which the new *rakes* and *compresses* are performed. Before making any other modification to the tree (such as a *link*, a *cut*, or another *expose*), however, we must restore the original contraction, or else we lose the guarantee that the individual components have logarithmic height.

To make it possible to restore the original contraction efficiently, we do the following. When the call to $expose(v, w)$ splits a cluster containing v or w, we only mark it as deleted; we actually retain the cluster in memory, including the pointers to its original children. We then build a temporary top tree on the unmarked root clusters. We mark all new clusters as temporary and store in the new root R a pointer to the root cluster R' of the original top tree.

Before performing an *expose*, *link*, or *cut*, we first check if the root R of each affected top tree is temporary. If it is, we *split* R and all descendents that are also temporary, and call *join* on all clusters of the original tree that have been marked as deleted. A pointer to the original root will be available at R. In accordance to the top tree interface, the *splits* must be performed top-down, and the *joins* bottom-up. This will restore the original top tree.

Note that this implementation of *expose* never changes the Euler tour representing the original contraction, since it will eventually be reused. This makes this approach much more efficient in practice than actually updating the original contraction, as suggested in the previous section.

It is convenient, however, to create a temporary Euler tour when building the temporary trees, since we have to make sure that the adjacency lists are sorted consistently with the original tree. But one does not need to create a temporary Euler tour for each level of the temporary contraction. It suffices to create a base tour only, and to gradually shortcut it as the contraction progresses. This tour can be discarded once the temporary top tree is built.

3.4 Alternative Design Choices

This section discusses some natural alternatives to some of the choices made in the design of our data structure.

3.4.1 No Circular Order

As already mentioned, defining a circular order of edges around each vertex makes the data structure more general, but it imposes what may seem to be excessively tight constraints on *rakes*. What would happen if adjacency lists were unordered? In other words, what if a leaf adjacent to a vertex v could be *raked* onto any other edge adjacent to v?

This would obviously help the round-based contraction scheme. If the contraction works in maximal rounds, its height will still be bounded by $O(\log n)$. We can show, however, that updating a contraction with these rules (after a *link* or *cut*) may take as much as $\Theta(\log^2 n)$ time. Consider a star, i.e., a tree with a distinguished *center* v and all other vertices directly connected to it. Any contraction of this tree will consist only of *rakes* around v (except possibly for the last move, which may be a *compress*). If the number of edges on a level is even, all edges will be paired up in the contraction; if the number is odd, exactly one edge will remain unmatched.

Take a star with exactly $n = 2^k + 1$ vertices, for some integer $k > 1$. It is easy to see that every level of any contraction of this tree will have an odd number of edges, and therefore exactly one edge will be unmatched.[6] We say that all ancestors of an unmatched cluster (excluding the cluster itself) are *tainted*: if the tree changes and the unmatched cluster participates in some other move, the ancestors will have to be deleted.

Consider a particular contraction of the star above. Let $\eta(i) = 2^{k-i} + 1$ be the number of nodes on the i-th level of the top tree (starting at level zero). On the each level, label the nodes from 1 to $\eta(i)$, from left to right. On the i-th level, let the unmatched node be the one labeled $\lceil \eta(i)/2^i \rceil = 2^{k-2i} + 1$. Figure 3.12 shows an example with $k = 7$. Note that this distribution is such that the unmatched vertices have few common ancestors.

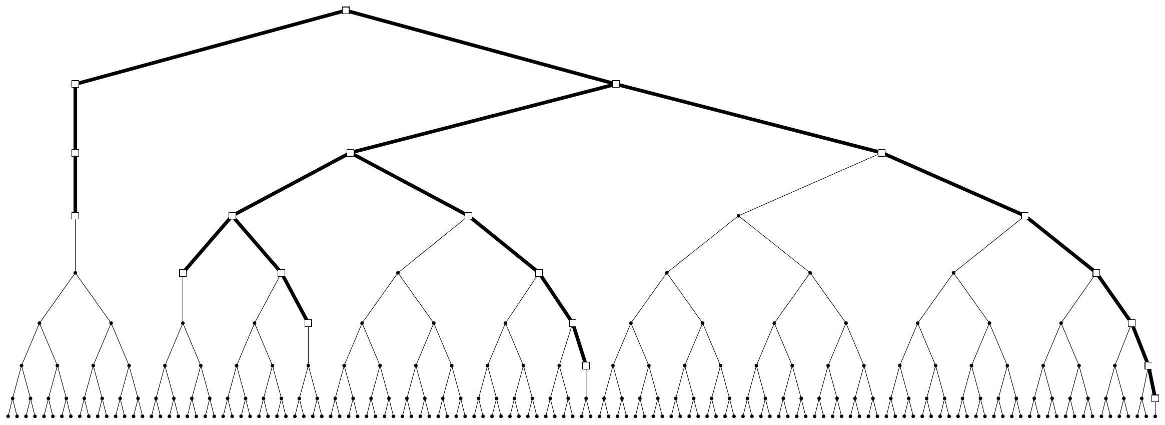


Figure 3.12: Why pairing up edges arbitrarily does not work. This top tree is a possible representation of a star with 129 edges. Tainted clusters (ancestors of unmatched clusters) are marked as hollow squares. They would all have to be *split* if an edge were removed from the original tree. In general, a top tree with n vertices may have $\Theta(\log^2 n)$ tainted clusters.

More precisely, let h be the height of the tree, i.e., the number of vertices on a path from a leaf to the root. Ignoring the root, we see that the unmatched node at level zero will have $h - 2$ tainted ancestors. The unmatched node at level 1 will have $h - 4$ tainted

[6]There is only one exception: the level immediately below the root.

ancestors (excluding the ones already tainted by level 0). In general, the unmatched node at level i will taint $h - 2(i+1)$ previously untainted clusters, for $i < h/2$. To simplify the analysis, we ignore the nodes tainted because of unmatched nodes on level $\lceil h/2 \rceil$ or above. The total number T_h of tainted clusters will be at least

$$T_h \geq \sum_{i=0}^{\lfloor (h-1)/2 \rfloor} (h - 2(i+1)).$$

When h is even, this translates into

$$T_h \quad \geq \quad \sum_{i=0}^{h/2-1} (h - 2(i+1)) = \frac{h^2}{2} - 2\sum_{i=1}^{h/2} i = \frac{h^2}{4} - \frac{h}{2}.$$

For h odd, the asymptotic bound is similar:

$$T_h \quad \geq \quad \sum_{i=0}^{(h-1)/2} (h - 2(i+1)) = \frac{h^2+h}{2} - 2\sum_{i=1}^{(h+1)/2} i = \frac{h^2}{4} - \frac{h}{2} + \frac{1}{4}.$$

Since $h = \Theta(\log n)$, this result implies that $\Omega(\log^2 n)$ clusters will be tainted in this case. When an edge is *cut* from the original tree, every level will have an even number of edges, which means that every originally unmatched edge will have to be matched, causing all tainted clusters to be *split*. We have thus proven the following:

Theorem 3 *A contraction scheme that requires maximality but has no ordering constraint needs at least* $\Omega(\log^2 n)$ *time to be updated in the worst case.*

Interestingly, the original top tree interface defined by Alstrup et al. [10] does not have any ordering constraint. In principle, any two edges adjacent to the same vertex can be combined, as long as the other endpoint of one of them has degree one. Both implementations they suggested for the top tree interface, however, do end up imposing an order on the adjacency lists: the direct (flawed) implementation in [34] does so explicitly; the implementation that uses topology trees does so implicitly, through ternarization.

3.4.2 Back Rakes

Eliminating the circular order would be an effective way of increasing the number of moves in each level of the contraction, but the previous section has shown that it poses problems if we still require maximality. A simpler, less drastic alternative would be to allow *back rakes*. As previously described, the contraction algorithm only allows a leaf edge to be *raked* onto its successor; we could allow it to be *raked* onto the predecessor as well. Although the case analysis would have to be redone to consider this case, this change seems to be small enough for us to assume that we would still be able to perform updates in $O(\log n)$ time. Its effect in practice would have to be tested. It is reasonable to expect that the average number of levels on a tree would decrease slightly, but the cost of processing each level could increase because new types of moves have to be tested. It is not clear what the balance between these two effects would be. Using a strict circular order (with forward *rakes* only) simplifies the analysis and is enough to guarantee a logarithmic worst-case performance.

3.4.3 Alternating Rounds

An early version of our data structure alternated between *rake* and *compress* rounds. Each round still had to be maximal, but only one kind of move could be performed. These constraints make the analysis of the update algorithm slightly simpler, at the expense of doubling the expected number of rounds necessary to perform a contraction. This has been done before: RC-trees alternate between *rake* and *compress* rounds (also to simplify the analysis). The analysis of RC-trees is further simplified by the fact that trees have bounded degree and that information is accumulated on vertices, not edges.

3.4.4 Randomization

Another reason why the analysis of RC-trees might be simpler than ours is the fact that RC-trees are randomized. When there are two conflicting moves within the same round, an implementation of our algorithm can arbitrarily pick either one. RC-trees, on the other hand, use consistent hash functions to make consistent choices even when the underlying tree changes. This makes the update algorithm simpler, but it ends up performing fewer moves per level that it could, which results in more rounds. Another undesirable feature is that the $O(\log n)$ bound on update times is randomized, not worst-case.

We note, however, that randomization is not a necessary feature of RC-trees. Inspired by our results on top trees, the creators of RC-trees managed to prove an $O(\log n)$ worst-case bound for updates on their data structure [2]. Instead of using the randomized oracle to decide which moves to make during an update, the worst-case version tries to imitate as many moves as possible from the original contraction. The analysis of the worst-case version of RC-trees is not much more complicated than that of the randomized version.

An obvious question regarding our data structure is whether randomization would help. In light of the finding regarding RC-trees, that does not seem to be the case. In fact, it might actually make the proof more complicated. Whereas RC-trees need randomization to deal with conflicting *compress* moves only (since *rakes* do not interfere with one another), our data structure would need randomization to organize *rakes* as well.

Chapter 4

Self-Adjusting Top Trees

This chapter presents *self-adjusting top trees*, a data structure that implements the top tree interface using path decomposition techniques similar to those used in Sleator and Tarjan's ST-trees. Self-adjusting top trees were first introduced by Tarjan and Werneck [57].

The chapter is organized as follows. Section 4.1 describes how a forest is represented by our data structure. Section 4.2 shows how queries and updates are handled. Section 4.3 establishes the $O(\log n)$ amortized time per operation. Section 4.4 suggests possible simplifications to the data structure and alternative design choices. Section 4.5 discusses the relationship between tree contraction and path decomposition. Final remarks are made in Section 4.6.

4.1 Representation

As in ST-trees, we will represent a partition of the tree into disjoint paths. Instead of making them vertex-disjoint, however, the paths will be *edge-disjoint*. More precisely, to represent a free tree we first pick a degree-one vertex as the *root* and direct all edges towards it. We call this (a directed tree whose root has degree one) a *unit tree*. We then partition the tree into non-crossing, edge-disjoint paths that begin at a leaf and end at another path. The only exception is the *root path* (or *exposed path*), which ends at the root.

Since we are supposed to implement the top tree interface, our goal is to create a cluster to represent the entire unit tree. Any internal vertex v of the root path P has exactly two neighbors on the path and zero or more *outer neighbors* (i.e., neighbors that do not belong to the root path). Since edges around a vertex are arranged in circular order, the list of arcs incident to v is divided by P into two (possibly empty) subsequences (see Figure 4.1). Each element of these subsequences is a unit tree rooted at v, and therefore can be recursively represented by a single cluster. Clusters in the same subsequence are progressively paired up to create a *rake tree*: its root represents the entire subsequence, leaves represent unit trees, and each internal node is the *rake* of the left onto the right child.

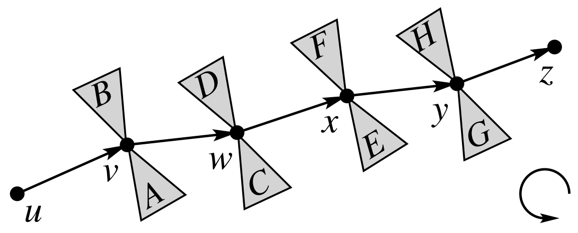


Figure 4.1: A unit tree rooted at z. The root path is $uvwxyz$. Triangles represent subtrees rooted at the root path.

We are now left with a path containing some k base clusters and at most $2k-2$ incident *outer clusters*, each representing a subtree. Empty subtrees will have no associated cluster. Ignoring the outer clusters, we could represent the path by a *compress tree*, a binary tree whose leaves are base clusters, and whose internal nodes represent *compresses* of adjacent clusters. Each node in the *compress* tree represents a subpath of the original path: leaves represent original edges and internal nodes represent nontrivial subpaths.

We deal with the outer clusters by *raking* them onto the root path. This is done as late as possible: an outer cluster incident to vertex v is *raked* just before v is *compressed.* In the data structure, it will become a *foster child* of the node representing *compress*(v); the two original children are *proper children.* The left foster child is *raked* onto the proper left child, and the right foster child onto the proper right child. The resulting clusters are then *compressed.*

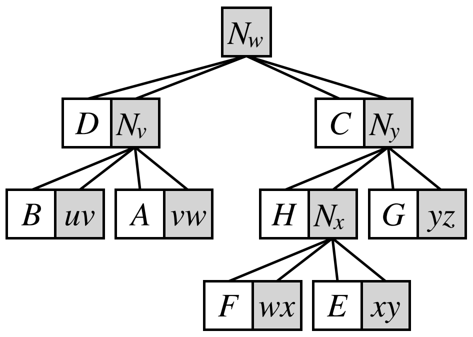


Figure 4.2: An augmented top tree representing the tree in Figure 4.1. Foster children are represented in white; all other nodes belong to the *compress* tree.

Figure 4.2 is a possible representation of the unit tree in Figure 4.1. Shaded nodes belong to the *compress* tree. Internal nodes are labeled by the vertices *compressed* (e.g., N_y represents *compress*(y)). Each internal node has up to four children and represents at most three clusters: two *rakes* (one for each foster child) and one *compress* (of the clusters generated by the *rakes*). Each foster child is shown in white and is adjacent to the corresponding proper child (the one onto which it is *raked*). Recall that a foster child is actually the root of a binary (*rake*) tree whose leaves represent unit trees. We call the representation in Figure 4.2 an *augmented top tree*, since its *compress* nodes have up to four children.[1]

Summing up, we represent a unit tree as follows:

1. Recursively compute clusters to represent each unit tree incident to the root path P.

2. Create *rake* trees to represent each contiguous sequence of unit trees incident to P.

3. Create a binary tree of *compress* nodes to represent the root path, with the *rake* trees appearing as foster children.

This method works for the entire tree, which is itself a unit tree. The end result is a hierarchy of alternating *rake* and *compress* trees.

[1]In fact, Phil Klein has suggested using the term "splice node" to refer to a *compress* node of an augmented top tree. Although this notation is not used here, it is equally valid.

For a better understanding of this representation, Figures 4.3, 4.4, and 4.5 show a complete example. The left part of Figure 4.3 shows a free tree, with edges ordered in counterclockwise around each vertex. To represent it, we first pick a degree-one vertex as the root and direct all edges towards it. In the example, z is the root. We then partition the tree into maximal non-crossing edge-disjoint paths, all starting at some leaf. The right part of Figure 4.3 shows a possible partition.

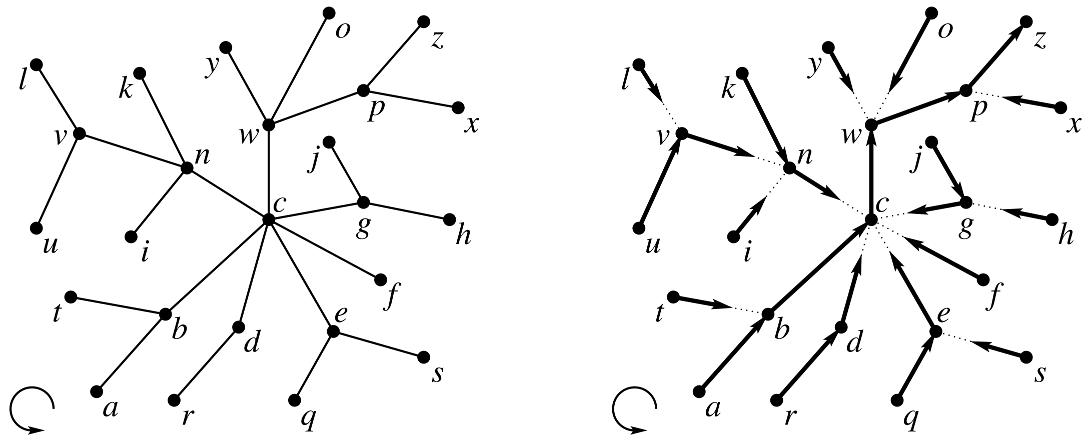


Figure 4.3: A complete example. On the left, the original free tree to be represented. On the left, the tree partitioned into vertex-disjoint paths and rooted at z. The root path is $abcwpz$.

An augmented top tree corresponding to this partition is shown in Figure 4.4. Base nodes are represented as shaded rectangles, *compress* nodes are white rectangles, and *rake* nodes are circles. When nodes appear paired up, the foster child is on the left, and the proper child on the right. Nodes that are not paired up are always proper children. Figure 4.5 shows the actual top tree corresponding to this augmented top tree. Note that the base nodes appear in the same order in both trees, and that new *rake* nodes need to be introduced to represent the moves between foster and proper children.

The root path in the example is $abcwpz$. It is represented by the top *compress* tree in Figure 4.4, which has N_b, N_c, N_w, and N_p as internal nodes, and ab, bc, cw, pw, and pz as leaves. Although the leaves represent the edges on the path, they need not appear in

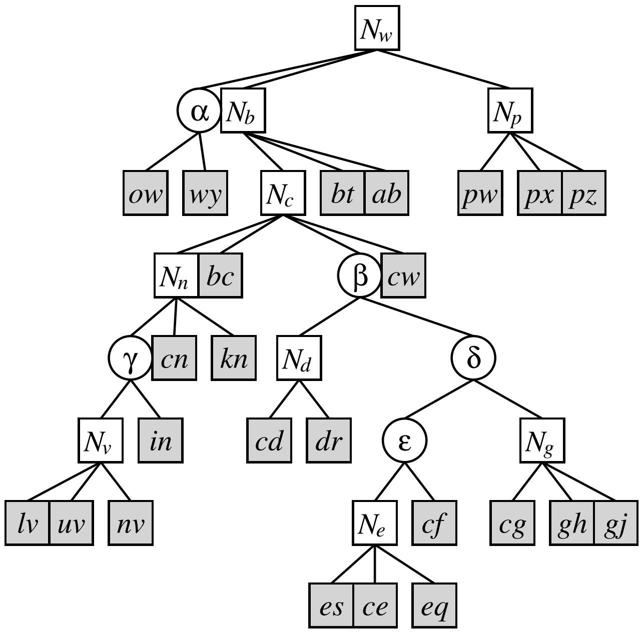


Figure 4.4: Augmented top tree corresponding to Figure 4.3. Base nodes are shaded rectangles, *compress* nodes are white rectangles, *rake* nodes are circles.

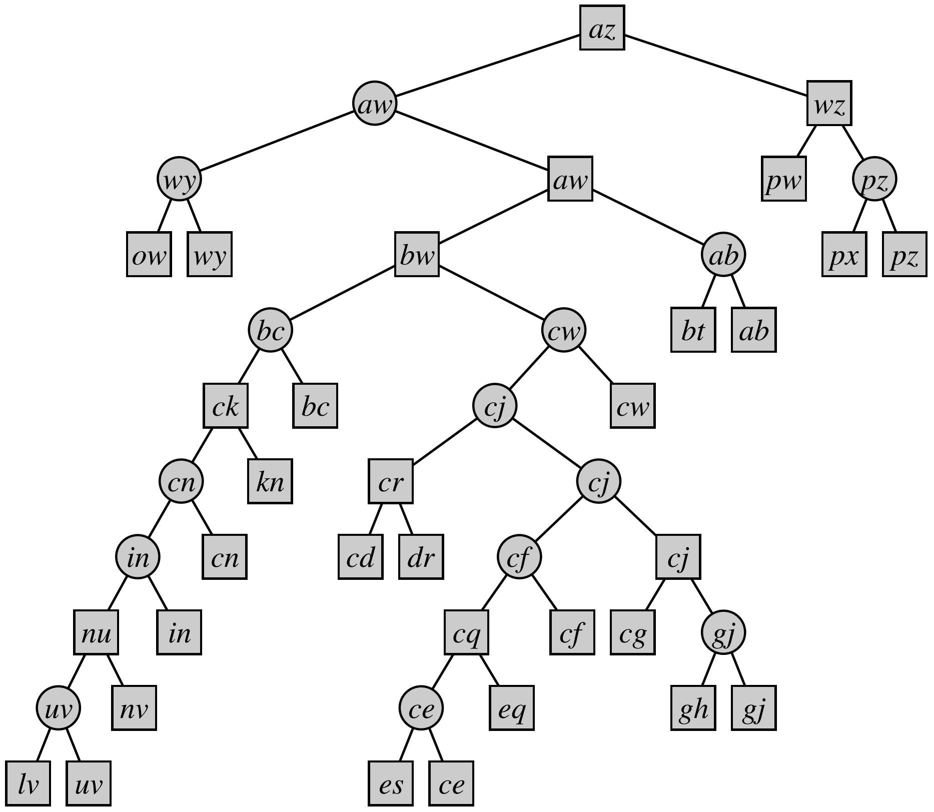


Figure 4.5: Top tree corresponding to the augmented top tree of Figure 4.4.

symmetric order, as Section 4.1.1 will explain. For example, edge ab appears before bc on the root path, but cluster bc appears to the left of cluster ab in the top tree.

The largest *rake* tree in the example has three internal nodes (β, δ, and ε) and four leaves (N_d, N_e, *cf*, and N_g). The leaves of this *rake* tree all represent unit trees that are rooted at c and occur between (b, c) and (c, w) in the circular order. (These are edges of the path that has c as an internal vertex.) The first (leftmost) unit tree is composed by edges (d, r) and (c, d); the rightmost contains (g, h), (g, j), and (c, g).

4.1.1 Order within Binary Trees

We have seen that a *rake* tree represents in symmetric order a sequence of clusters that appear consecutively around the same vertex. The leaves of a *compress* tree represent the edges of a path and in principle could also appear in symmetric order. To handle path reversals efficiently, however, we use a more relaxed condition. Given a node representing *compress*(v) with endpoints u and w, one of its subtrees must represent the path $u \cdots v$, and the other $v \cdots w$. Left and right subtrees can be interchanged freely. The "correct" order among children can be retrieved from the cluster endpoints.

ST-trees use a similar technique to support the *evert* (change root) operation, with a reverse bit used to retrieve the correct order when necessary, as mentioned in Section 2.1.4. This may seem less costly than our approach, since they use just one bit per node, whereas we use two words (one for each endpoint). However, endpoints must be kept in the top tree clusters anyway, so the information is essentially free.[2]

4.1.2 Handles

Some of the external top tree operations (*link* and *expose*) are defined in terms of vertices, but the top tree itself is a hierarchy of clusters and nodes, which can be viewed as edges or paths—not vertices. Therefore, we associate with each vertex v a *handle* N_v. If the degree

[2]Of course, one could argue that "necessary features" such as this may make top trees less than ideal for certain applications. There will always be some trade-off between ease of use and efficiency, but here the cost should not be too high.

of v is at least two, N_v is the node representing *compress*(v). If the degree is one, N_v is the topmost non-*rake* node that has v as an endpoint, which is either the root of the entire tree or a leaf of a *rake* tree. Isolated vertices have no handle. A node may be the handle of as many as three vertices. In Figure 4.2, for example, the root (N_w) represents both the path from u to z and *compress*(w); therefore, it is the handle of u, w, and z. The map from vertices to handles is maintained explicitly.

Note that this definition of handles differs from the one used in contraction-based top trees (see Section 3.3.4). In that case, handles refer to clusters (or, rather, arcs) at the base level, since the update procedure always works bottom-up. In both data structures, however, handles have the same purpose: mapping each vertex to a cluster containing it.

4.2 Updates

Before operating on a path, the top tree interface mandates that we first *expose* it, i.e., make it represented at the root node. The representation described in Section 4.1 requires both endpoints of the root path to have degree one. To handle an arbitrary path $v \cdots w$, we first pick a root path that contains $v \cdots w$ as a subpath, then we temporarily convert up to two *compress* clusters into *rake* clusters. We call the first step a *soft expose* of vertices v and w, and the second a *hard expose* of the path $v \cdots w$. We discuss each in turn, in Sections 4.2.1 and 4.2.2. We then detail how to implement *cut* in Section 4.2.3 and *link* in Section 4.2.4. Additional implementation issues are discussed in Section 4.2.5.

4.2.1 Soft Expose

The outcome of *soft_expose*(v, w) depends on how v and w are related to each other. If the vertices are isolated, nothing is done. If $v = w$ or v and w are in different components, N_v (v's handle) and N_w (w's handle) are simply brought to the root of their components.

The interesting case happens when v and w are different vertices in the same component: in this case, *soft_expose*(v, w) ensures that a cluster vw (representing the path $v \cdots w$) exists and is close to the root of the top tree. It works by first making N_w the root, then bringing

N_v as close to the root as possible (preserving N_w). When both v and w have degree two or greater, all three nodes (N_w, N_v, and vw) are different, and the outcome is the one depicted in Figure 4.6. If only one of the endpoints has degree one, we will have $N_v = N_w$; if both have degree one, we will have $N_v = N_w = vw$. To simplify hard expose, we require *soft_expose* to make both N_v and vw right children (unless they coincide with the root N_w). Note that this can always be accomplished, since the proper children of a *compress* node can be exchanged freely.

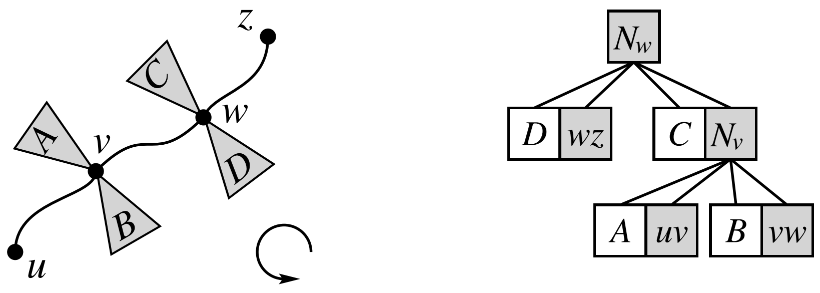


Figure 4.6: Configuration after *soft_expose*(v, w). This is the most general case, in which both v and w have degree at least two.

The *soft_expose* operation uses the same basic tools as the amortized version of ST-trees [52]: *splay* and *splice*. We discuss each in turn.

Splaying

Splaying [52] is a heuristic for rebalancing binary trees using rotations. After a node x is accessed, it is rotated up to the root. The precise nature of each rotation depends on the relative positions of x, its current parent p, and its current grandparent g. If x and p are both right (or both left) children (*zig-zig* case), we first rotate edge (p, g), then (x, p). If the edges alternate (*zig-zag* case) we rotate (x, p) first, then (x, g). When p is the root (*zig* case), we just rotate (x, p).

In self-adjusting top trees, rotations (and splaying) happen only within individual *rake* or *compress* trees—we never splay across different subtrees. We therefore perform *guarded* splays, which stop when x, the node being splayed, becomes the child of a reference node (the *guard*). An ordinary splay can be thought of as guarded by *null*.

As described, the rules for splaying apply to binary trees. However, *compress* trees have internal nodes with up to four children. From the point of view of the splaying rules, only *proper children* are considered. As shown in Figure 4.7, foster children are not affected by rotations: they always keep their original parents. The proper siblings they are *raked* onto may change, however.

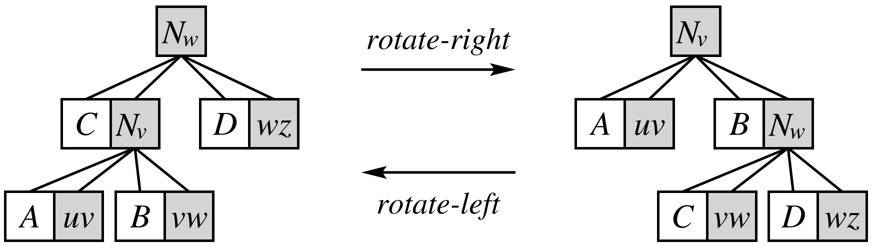


Figure 4.7: Rotations in *compress* trees. Note that foster children always preserve their original parents.

Splaying requires the left and right children of all nodes visited to appear in a consistent (symmetric) order. In general, we cannot assume that this is true for *compress* trees, since the children of *compress* nodes can be swapped freely. Therefore, before splaying on a node N we must *rectify* all *compress* nodes on the path from N to the root of its top tree (*rake* nodes always have the correct order). If a node X has parent N_y and grandparent N_z, then X, N_y, and N_z must form a zig-zag if and only if the endpoints of X are y and z. We ensure this by flipping the children of N_y appropriately. To preserve the circular order when proper children are flipped, we flip the foster children as well. Each *compress* node visited has two proper children, each representing a path. Rectification ensures that the path that is farthest from the root of the corresponding unit tree is represented at the left child. Rectification is performed in a top-down fashion.

We note that splaying is performed for balancing purposes only. It changes the order in which different moves (*rake* and *compress*) of the same type occur, but preserves the original partition into paths. In both top trees in Figure 4.7, the root node represents the path $uvwz$. On the left, vertex v is *compressed* before w; on the right, w is *compressed* before v.

Splice

The operation that changes the partition of the original tree into paths is *splice*. A vertex v that is internal to a path partitions this path into two segments. Splice replaces the segment that is farthest from the root with an *outer path* incident to v (i.e., with the root path of a unit tree rooted at v). In Figure 4.8, $x \cdots v$ is replaced by $y \cdots v$ on the exposed path.

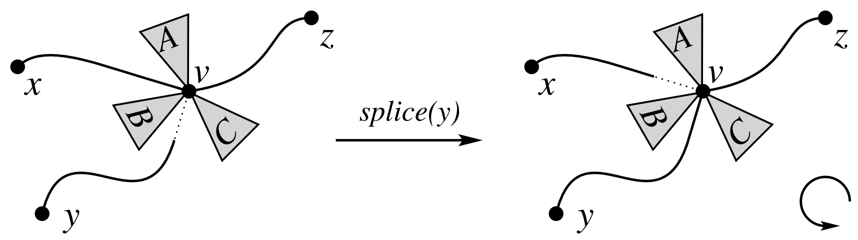


Figure 4.8: Splice: $y \cdots v \cdots z$ replaces $x \cdots v \cdots z$ as the exposed path. Triangles represent subtrees incident to v, and curved lines represent paths; subtrees incident to these paths are omitted.

Figure 4.9 shows a possible configuration of the corresponding augmented top trees. The original proper children of N_v (v's handle) are vx and vz, representing $x \cdots v$ and $v \cdots z$. We shall see that splices only occur after a series of local splays (within *compress* and *rake* trees). As a result, N_v will be the root of a *compress* tree, and there will be at most two *rake* nodes between vy, which represents the subpath we want to expose, and N_v. Splice makes vy the left child of N_v and incorporates the former left child (vx) into a *rake* tree, where it appears between A and B as required by the circular order.

Figures 4.8 and 4.9 represent the most general out of several possible cases for splice. The precise outcome depends on which foster child of N_v contains vy (the subpath to be exposed) and on whether some of the *rake* subtrees (A, B, or C) are absent. Figuring out

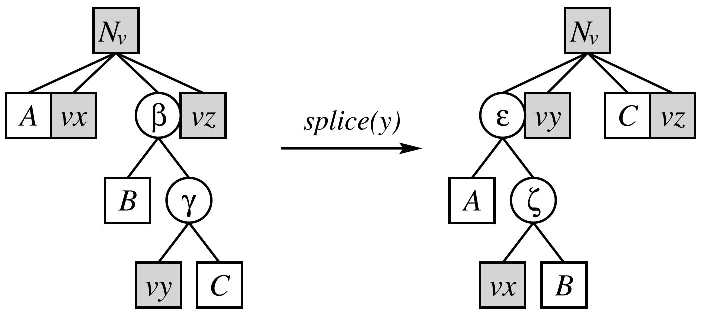


Figure 4.9: Splice: Augmented top trees corresponding to Figure 4.8. Circles represent *rake* nodes.

what to do in each situation is simple: one must always replace the left child of N_v (which, after rectification, represents the subpath that is farthest from the root of the unit tree) while ensuring that the circular order of the up to six relevant subtrees rooted at v (denoted by A, vx, B, vy, C, and vz in Figure 4.9) is preserved.

Exposing the Target

Now that we have the necessary building blocks, we return to the implementation of *soft_expose*(v, w). Its first step is to expose the target vertex w, making its handle N_w the root node.[3] The function starts from N_w itself, and works in three passes:

1. (Local Splay) Splay within each *compress* and *rake* subtree on the top tree path from N_w to the root.

2. (Splice) Perform a series of splices from N_w to the root, making N_w part of the topmost *compress* subtree.

3. (Global Splay) Splay on N_w, making it the root of the entire top tree.

We call this sequence of operations a *generalized splay.*

The first pass is divided into several subpasses, each starting from a different *compress* tree. Let N be a node of this tree (initially, $N = N_w$). Splay on it, making N the root of its *compress* tree and a leaf of the *rake* tree immediately above. If the *rake* tree contains

[3]We refer to w as the "target" for convenience only; the path $v \cdots w$ is actually undirected.

other nodes besides N, splay on N's parent, P, within the *rake* tree. If N ends up with a new parent P', splay on P' with P as a guard. This concludes the subpass.

The left of Figure 4.9 shows a possible configuration after a subpass associated with node vy. This node becomes the root of a *compress* tree (not shown), and between itself and the closest *compress* ancestor (N_v), there are at most two *rake* nodes (γ and β). In fact, we splay twice on the *rake* tree precisely to divide this tree into three subsequences: B, vy, and C.

After the first pass is completed, every node on the path (in the top tree) between N_w and the root will be as close as possible to the root of the corresponding subtrees (*rake* trees or *compress* trees). This allows us to perform a series of splices, the second pass of the algorithm outlined above. As a result, w will become part of the root path, and N_w part of the topmost *compress* tree.

At this point, we can execute the third pass of the algorithm, global splay. It consists of splaying within the topmost *compress* tree to make N_w its root, and therefore the root of the entire top tree.

Although the algorithm is easier to analyze if we think of it as having three passes, in our implementation we actually perform the first two passes simultaneously. Moreover, to avoid splaying twice within each *rake* tree, we actually perform a special *splaying split* to obtain the three subsequences mentioned above (B, vy, and C). The procedure is very similar to splaying on vy directly, but it is not exactly the same. In general, splaying on vy can make it an internal node of the *rake* tree, and some *rake* nodes may become leaves. This is not allowed by our representation. A splaying split does splay on vy, but in the process removes from the tree its immediate predecessor and its immediate successor in symmetric order (which are both *rake* nodes). These nodes are reinserted into the top tree by splice, which happens immediately after the splaying split.

Exposing the Source

Having seen how to expose the target w, we now consider the source v.

After the target is exposed, N_w will be the root. If v is an endpoint of N_w or if N_w represents *compress*(v) (in which case w must be an endpoint of N_w), we are done: N_w is v's handle as well. Otherwise, we must bring v's handle (N_v) as close to the root as possible.

The basic idea is to apply to N_v a three-pass procedure similar to the one applied to N_w. While doing so, we must ensure that the root of the top tree will remain the handle of w. The exact procedure depends on the degree of w.

If w has degree one, it will be one of the endpoints N_w, the root of the top tree. To expose v, we first make sure that the right child of N_w has w as an endpoint; if it does not, it suffices to flip the children of N_w. This guarantees that w will remain part of the root path even after a splice, which always removes left children. We then apply to N_v a generalized splay, the three-pass procedure previously applied to N_w. This will make N_v the root, with w guaranteed to be one of its endpoints: N_v and N_w will coincide.

If w has degree two or greater, then N_w will represent *compress*(w). To expose v, we apply to N_v a generalized splay, but guarded by N_w: every splay in the procedure is guarded by N_w. This ensures that no node will replace N_w at the root, so either N_v will end up as N_w's child (when v has degree at least two), or v will become an endpoint of N_w (and $N_w = N_v$ will be the root).

To follow the specification of *soft_expose*, we may need to flip the children of N_w and N_v. If $N_v \neq N_w$, N_v must be N_w's right child; if the node representing $v \cdots w$ is not N_v, it must be N_v's right child. If v and w are in different components, the generalized splay will end up exposing N_v as if it were the target, as required by the specification.

4.2.2 Hard Expose

In general, *soft_expose*(v, w) does not make v and w the endpoints of the root node. Instead, as Figure 4.6 shows, the root node will represent some path $u \cdots z$, with vw as the rightmost grandchild. To fix this, *hard expose* temporarily converts to *rake* the (at most two) *compress* ancestors of vw. In Figure 4.6, N_v and N_w would be affected. Before another pair of vertices is exposed, these modifications are undone to bring the tree back to its "normalized" form.

4.2.3 Cuts

To cut an edge (v, w), we first execute *soft_expose*(v, w), making N_w the root. In the general case, both v and w have degree at least two. As the tree on the left of Figure 4.10 shows, N_w's right child will be N_v, and N_v's right child will be the base node representing (v, w).

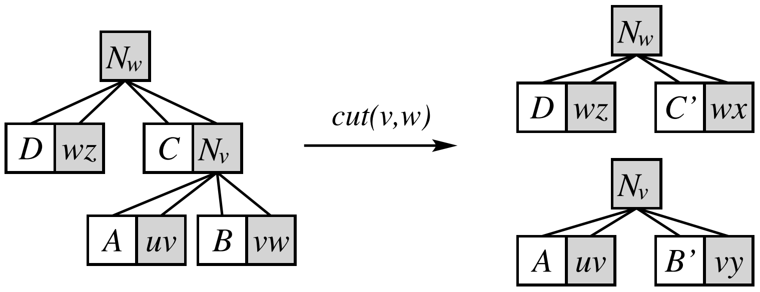


Figure 4.10: Cutting edge (v, w). The augmented top tree on the left is the one obtained after *soft_expose*(v, w), so vw is a base cluster. Clusters A, B, C and D are foster subtrees. During the *cut*, B is partitioned into vy (the rightmost leaf of the *rake* tree) and B'. Similarly, C is partitioned into wx and C'. Note that the circular order is preserved in the new trees.

We must destroy the base node and reorganize the remaining nodes into two valid top trees. In the original augmented top tree, N_w represents a unit tree with root path $u \cdots z$. The subtree rooted at N_v (the right child of N_w) represents a unit tree rooted at w with (v, w) as the only edge incident to the root. If we remove the link between N_v and N_w, N_w will be the root of a tree containing only the vertices in w's component. Similarly, if we remove the right child of N_v, only vertices in v's component will remain.

In both cases, the original right child must be replaced. First consider how to replace the right child of N_w, which is N_v. To preserve the circular order around w, the replacement must be either the immediate successor of vw around w (the leftmost leaf of the left foster subtree of N_w, denoted by D in Figure 4.10) or the immediate predecessor (the rightmost leaf of the right foster subtree of N_w, denoted by C). To extract the appropriate leaf, we simply splay on its parent. If N_w has no foster child, we delete N_w and make its left proper child the new root.

The right child of N_v (which is vw itself) can be replaced in a similar fashion, but considering the circular order around v instead: either the leftmost leaf of A or the rightmost leaf of B can be used.

4.2.4 Links

To insert an edge (v, w) as the successor of path $a \cdots v$ around v and of path $b \cdots w$ around w, we first perform *soft_expose*(a, v) and *soft_expose*(b, w). Of course, av or bw can be single edges instead of paths. If both v and w have degree greater than one, we will have the two augmented top trees shown on the left of Figure 4.11. To link them, we do the opposite of *cut*: we first replace the right child of N_v with vw, making the original right child (N_a) the rightmost leaf of the right foster subtree of N_v. (If this foster subtree is originally non-empty, a new *rake* node must be created.) Then we do the same for N_w, making N_v its new right child.

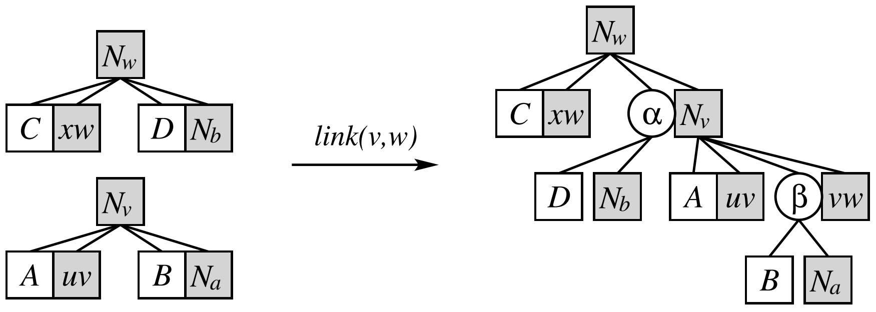


Figure 4.11: Linking v and w. The augmented top trees on the left are the result of calls to *soft_expose*(a, v) and *soft_expose*(b, w). Nodes are rearranged so as to make vw (the new edge) the successor of path $a \cdots v$ around v and of path $b \cdots w$ around w.

The case represented in Figure 4.11 is the most general. We rearrange vertices in a similar way when v or w have degree one or zero. In particular, when v has degree one before the *link*, it is an endpoint of N_v; to add (v, w) to the tree, we create a new *compress* node with the old N_v and the base node representing (v, w) as children. A new *compress* node is also necessary when w has degree one.

4.2.5 Implementation Issues

Handling Data

So far, we have discussed only structural changes to the top trees. To update the values in each cluster, we need the user-defined functions *create*, *destroy*, *join*, and *split*. Rotations and splices can be easily expressed in terms of a series of *splits* and *joins*. However, since the top tree is modified in a bottom-up fashion, these functions cannot be called as we go: they can only be applied to root clusters. Instead, before exposing a vertex, we mark all affected clusters, then *split* them in a top-down fashion. (We can actually do this together with the rectification pass.) Only then do we perform all structural modifications. A simple recursive function unmarks all clusters and calls *join* in a bottom-up fashion once all the structural operations are performed.

Non-local Search

As mentioned in Section 2.2.4, the top tree interface can be augmented to include the *select* operation, used to guide a simulated binary search on the top tree. The actual running time of the binary search is proportional to the depth (in the top tree) of the last cluster visited. Let this cluster be C. Although its depth can be linear in the self-adjusting implementation, the *amortized cost* of the binary search will be logarithmic, as long as it is immediately followed by a generalized splay on C. Splaying will amortize the cost appropriately.

Nodes and Clusters

Strictly speaking, our representation does not implement top trees directly. We have nodes with up to four children, whereas top trees are binary. For a direct implementation, it suffices to replace each *compress* node by the three corresponding clusters (one *compress*, two *rakes*), as shown in Figures 4.3 and 4.4. Unfortunately, splay and splice would become considerably more complex, since they would have to account for the fact that *compress* trees now have interspersed *rake* nodes.

4.3 Analysis

The run-time analysis of the routines to update the data structure does not consider the augmented top tree itself but rather an equivalent *phantom tree*. In the augmented top tree, *compress* nodes have up to four children; in the phantom tree, up to three: left, middle, and right. To convert a four-child top tree node to a phantom tree node, we create an *articulation node* (the new middle child) and make it the parent of the original foster children. The articulation node is inserted only when there are two foster children. Figure 4.12 shows the phantom trees corresponding to the augmented top trees in Figure 4.9.

While a tree with n vertices may be represented by augmented top trees with various numbers of nodes, phantom trees will have exactly $n-1$ nodes; this greatly simplifies the analysis. Phantom trees are used in the analysis only; they do not appear in the implementation.

We extend Sleator and Tarjan's analysis of ST-trees [52]. The *rank* of a node N is defined as $r(N) = \log s(N)$, where $s(N)$, the *size* of N, is the number of nodes descending from N in the phantom tree. Note that the rank is at most $\log n$. The *potential* of the phantom tree is defined as q times the sum of the ranks of all nodes, where q is a constant to be chosen later. The *amortized cost* of the i-th operation in a sequence is defined as $a_i = c_i + \phi_{i+1} - \phi_i$, where c_i is the *actual cost* of the operation, and ϕ_i and ϕ_{i+1} are the potentials before and after it is performed. A bound on the total amortized time translates into a bound on the actual time [55].

In general, the operations we perform within a pass take an *active node* and move it upward in the tree. The node that is active may even change during the process, in particular during a splice and between local splays. The depth of the active node (regardless of whether it is a different node or not) is guaranteed to decrease from one step to the next, however. Each basic operation (rotation, double rotation, or splice) deals with a constant number of nodes, and therefore takes constant time. We define the actual cost of the i-th operation (c_i) as the amount by which the depth of the active node is reduced.

Rotations within *rake* and *compress* trees follow Sleator and Tarjan's analysis [52].

Their Access Lemma states that the amortized time for a zig-zig or a zig-zag on a node N is $3q(r'(N) - r(N))$, where $r(N)$ and $r'(N)$ denote the rank of N before and after the operation. Moreover, the amortized cost of a zig is $3q(r'(N) - r(N)) + 1$. Sleator and Tarjan use $q = 1$ in the analysis of standard splay trees and $q = 2$ when analyzing ST-trees, but any constant $q \geq 1$ can be used.

We need to obtain a similar result for splices. Figure 4.12 shows how splices work on phantom trees: just as in Figure 4.9, with articulation nodes (α and δ) added where necessary. The active node is vy before the operation, and N_v after. We can prove the following:

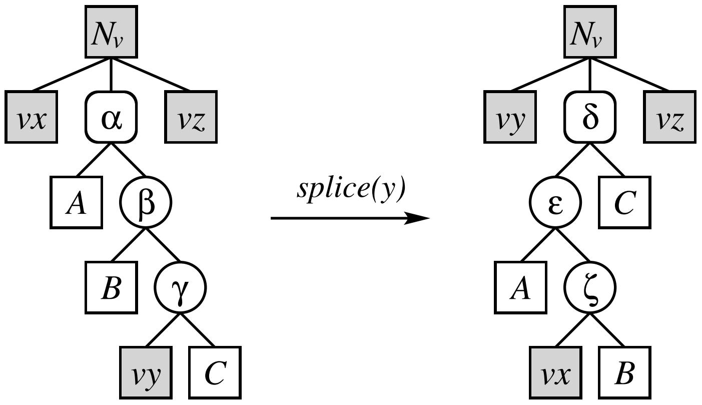


Figure 4.12: Splicing on the phantom tree corresponding to the tree on Figure 4.9, with α and δ added as articulation nodes.

Lemma 12 *The amortized cost of a splice is at most $3q(r'(N') - r(N)) + 4$, where N is the active node before the operation, and N' is the active nodes afterwards.*

Proof. The actual cost of the operation is at most 4, an upper bound on the amount by which the depth of the active node is reduced in the phantom tree. See Figure 4.12: the active node is $N = vy$ before the splice, and $N' = N_v$ after. The only nodes whose ranks change are those labeled with Greek letters in the figure; all others will keep the exact same set of descendants. All three affected nodes on the left are ancestors of the original active node $N = vy$, so their combined rank is at least $3r(N)$; the affected nodes on the right are all descendants of the final active node $N' = N_v$, which means their combined rank is at most $3r'(N')$. The amortized cost a of the operation is therefore

$$a = c + \phi' - \phi \leq 4 + q(3r'(N')) - q(3r(N)),$$

as claimed. Note that the contribution of all other nodes (besides the ones labeled with Greek letters) to the potential can be ignored, since their ranks are the same before and after the operation.

A similar analysis holds if there are fewer than three nodes between N and N'; the only difference is in the additive constant: instead of 4, we will have 3, 2, or 1. The case shown in Figure 4.12 is the most expensive. $\square$

We can now bound the amortized cost of *soft_expose*(v, w). It is enough to bound the time to expose the target vertex w; the same bound applies to the source v. We analyze each pass (local splays, splices, and global splay) in turn.

Consider the first pass. If k is the number of *compress* trees on the path from N_w to the root of the top tree, we will splay within k *compress* trees and within up to $k - 1$ *rake* trees (twice in each such *rake* tree). We can analyze this as two subpasses that go strictly upwards. The first subpass accounts for all rotations within *compress* trees and for the first splay within each *rake* tree; the second subpass accounts for the second splay within each *rake* tree.

Each subpass can be further divided into steps, which are either rotations (zigs) or double rotations (zig-zigs or zig-zags). Let s be the total number of steps in a subpass. Also let N_i be the active node before step i and N'_i be the active node immediately after step i. The amortized cost of the first subpass is

$$A_1 = 3q\sum_{i=1}^{s}(r'(N'_i) - r(N_i)) + 2k - 1.$$

Note that the $2k - 1$ term accounts for all zigs in the subpass (at most k within *compress* trees and $k - 1$ within *rake* trees). Because the pass only moves up the tree, we know that

$r(N_{i+1}) \geq r'(N'_i)$. Using this fact and defining $r(N_{s+1})$ to be equal to $r(N'_s)$, we have:

$$A_1 \leq 3q \sum_{i=1}^{s} (r(N_{i+1}) - r(N_i)) + 2k - 1.$$

Because the summation telescopes, we have that

$$A_1 \leq 3q(r(N_{s+1}) - r(N_1)) + 2k - 1 \leq 3q \log n + 2k - 1.$$

The second inequality uses the fact that no rank is greater than $\log n$, where n is the total number of nodes in the tree.

The second subpass (corresponding to the second splay within each *rake* tree) can be analyzed in a similar fashion. The only difference is that it splays in at most $k-1$ trees, making the amortized cost at most $3q \log n + k - 1$.

Therefore, the total amortized cost of the first pass of the algorithm is $6q \log n + 3k - 2$.

The second pass performs $k-1$ splices. From Lemma 12, the total amortized cost is at most $3q \log n + 4(k-1)$. Once again, the sum of ranks telescopes.

Considering just the first two passes, the total amortized cost of the procedure is $9q \log n + 7k - 6$. This would be $O(\log n)$, except for the $7k$ term, which can be up to $\Theta(n)$. The third pass of the algorithm (global splay) will pay for this extra term.

The global splay reduces the depth of the active node from $k-1$ to 0 with $k-1$ rotations within the same *compress* tree. The total amortized cost of the step is $3q \log n + 1$. Our potential function charges q time units per rotation, but the actual cost is one, which leaves us $(q-1)(k-1)$ "unused" units. Setting $q = 8$, we will be only one unit short of fully paying for the extra $7k - 6$ units spent on the first two passes.

Adding up the amortized costs of all three passes, we conclude that the total cost of exposing the target vertex is bounded by $12q \log n + 2 = 96 \log n + 2 = O(\log n)$. The same applies to the source.

We claim both *link* and *cut* also take $O(\log n)$ amortized time.

The *link* operation starts with a call *soft_expose*, which takes $O(\log n)$ amortized time. It then performs a constant number of pointer modifications. Because all affected nodes have constant depth (i.e., they are within a constant distance from the root), each of them cannot increase the overall potential by more than $O(\log n)$.

The *cut* operation also starts with a *soft_expose* and modifies node pointers that are close to the root, with the same bounds as *link*. Moreover, a *cut* also performs one additional splay within a foster subtree of each new root. These splays may visit nodes of arbitrary depth, but, because they are just standard splays, their total amortized time is $O(\log n)$ as well.

We have thus proved the following:

Theorem 4 *Self-adjusting top trees support* link*,* cut*, and* expose *in $O(\log n)$ amortized time.*

4.4 Alternative Representations

4.4.1 Possible Simplifications

Although self-adjusting top trees have some features in common with ST-trees, they are much more general: they support subtree operations on trees of unbounded degree, ordered incidence lists, and unrooted trees (without the need for the *evert* operation). Many applications, however, have no need for one or more of these extra abilities. In fact, using such a general data structure could have a negative effect on performance. This section discusses how our data structure can be simplified when its full power is not needed.

Unordered Adjacency Lists

We first consider applications in which the order among the edges incident to a vertex is irrelevant. In such cases, instead of maintaining two foster children per *compress* node, we can replace them with a single middle child, in a manner similar to phantom trees. This middle child must be interpreted as being *raked* onto either proper child. Not only does the representation itself get simpler, but it also becomes easier to update: there are fewer cases (and nodes) to handle when performing *splices*, *cuts*, and *links*.

Trivial Rakes

Further simplification is possible for applications in which *rakes* have no effect on the target cluster, i.e., when the data in the parent cluster is a copy of that in the target. This is what happens in applications that deal exclusively with paths, not trees or subtrees. In such situations, an edge is only *raked* when we know it does not belong to the path we are interested in, and therefore we can ignore the information it holds. Then we can completely eliminate *rake* trees and link the root node of each *compress* tree directly to the *compress* node above, mimicking the dashed edges of ST-trees [51].

Whereas ST-trees do not need pointers from a node to its dashed children, however, our data structure needs at least one such pointer. The reason is that, during a *cut* operation, we might need to replace a proper child of a node with one of its foster children. Since there is no circular order, any foster child is good enough. Therefore, it suffices to keep a pointer from a node to one of its foster children, as long as the foster children are maintained in a circular list. The list must be doubly-linked, because it must allow arbitrary deletions during splices. Whenever a foster child is "promoted" to be a proper child, it is removed from its circular list and replaced by its successor.

Rooted Trees

Another special case is that of rooted trees. ST-trees assume the underlying trees are rooted; to handle free trees, there is a special operation to change the root (*evert*). Although our data structure already incorporates *evert* into *expose*, it does get slightly simpler when roots are fixed, since the rectification step becomes unnecessary.

4.4.2 Unit Trees

One aspect of our representation that seems arbitrary is the notion of unit trees. Recall that the root path must begin and end at vertices of degree one. ST-trees have no such constraint. Unit trees are the only reason why we must have *hard expose*, which temporarily converts up to two *compress* clusters into *rake* clusters in order to make a subpath of a root path exposed. If we did not have the notion of rooted trees, a single *expose* operation would suffice. Why did we choose to use unit trees?

We chose it mainly for symmetry, which in turn results in fewer special cases. Consider the top tree in Figure 4.2, which represents the tree in Figure 4.1. Every node of the *compress* tree is matched with a foster sibling, with the exception of the root. Every cluster with a foster sibling will be the target of a *rake* inside this tree.

Now consider what would happen if we allowed *compress* trees to be more general: instead of representing paths that start at a degree-one vertex, it could represent paths starting at arbitrary vertices. In other words, consider a tree similar to that of Figure 4.1, but with an extra subtree U rooted at u (besides the subtrees A to H, already drawn). How would U be represented within the binary tree? Clearly, it must be *raked* onto some cluster. The obvious alternatives are to *rake* it onto the base cluster uv or maybe onto the root cluster of the top tree—the one representing the entire uv path (represented as N_w in the top tree). In either case, a new *rake* cluster would have to be incorporated into the *compress* tree. Every element of the update of algorithm (in particular, splices and rotations) would have to account for this special node.

Requiring each path to start at a degree-one node eliminates this special case in all but one *compress* tree, and that is why we chose this representation. The *compress* tree that needs special treatment in our representation is the topmost one, which represents the exposed path. Treating this one special case is the purpose of *hard expose.*

4.5 Path Decomposition and Tree Contraction

Our data structure demonstrates that the two main approaches used to represent dynamic trees are, in a sense, equivalent. ST-trees represent a partition of the trees into disjoint paths. Topology trees, RC-trees, and top trees are based on tree contraction. Frederickson [25] noticed that partitions and contractions have similarities, and Alstrup et al. [10] even showed that topology trees can be implemented using ST-trees. The transformation is far from direct, however, as the authors themselves observe. In contrast, there is no need to "map" our data structure from one framework to the other. One can simply interpret it in two different ways.

From the point of view of the interface, our data structure is a pure top tree: a single tree with internal nodes representing either *rakes* or *compresses*, and leaves representing base clusters. As such, it is as powerful (and as general) as any other implementation of top trees.

From the point of view of the update algorithms (*expose*, *link*, and *cut*), our data structure follows the path decomposition framework; it is interpreted not as a single tree, but as a hierarchy of binary trees (*rake* trees or *compress* trees). This makes the implementation almost as simple as the implementation of ST-trees.

The correspondence between tree contraction and edge-disjoint path decomposition is not always that obvious, but it always exists. Any valid tree contraction can be associated with a valid path decomposition, and vice-versa. We consider each direction in turn.

4.5.1 Contraction to Decomposition

First, note that any sequence of *rakes* and *compresses* can be translated into a unique partition of the original tree into edge-disjoint paths. Two edges a and b will belong to the same solid path if and only if there is a node in the top tree with endpoints s and t such that both a and b belong to the (unique) path between s and t in the original tree. Intuitively, solid paths are "grown" by *compress* moves. A *compress* node C with children A and B indicates that the path represented by A and the path represented by B belong to the same solid path.

The simplest way to identify solid paths, however, is by looking at the *rake* moves. A *rake* cluster indicates that one path (the one represented by the cluster being *raked*) will stop growing. This implies that the solid paths in which the original tree is partitioned are exactly the paths represented by all the left children of *rake* clusters, plus the root of the entire top tree (the only path that is not *raked*).

In the example in Figure 2.6, the solid paths are ac, dg, ij, lm, fg, mo, bg, hi, and en

(the root). This being a very small example, only two solid paths (bg and en) have more than one edge.

A larger example is the top tree presented in Figure 4.5. By taking the root of the tree and the left children of *rake* nodes, we see that the solid paths are: ow, wy, lv, nu, in, ck, cr, cj, es, cq, cf, gh, bt, az (the root), and px (the nodes are listed in symmetric order). These are exactly the solid paths depicted in Figure 4.3.

4.5.2 Decomposition to Contraction

If every contraction can be mapped into a partition, the converse is also true: any partition into paths can be translated into a sequence of *rakes* and *compresses* (although not necessarily unique).

In the representation suggested in this chapter, each solid path is represented as a *compress* tree. Despite the name, these trees have more than *compress* nodes: there may be *rake* nodes amid them. Furthermore, our representation imposes a particular set of constraints on these *rake* trees. We know, for instance, that there can be no more than one *rake* node between a *compress* node (or a base node) and the *compress* node immediately above it in the same *compress* tree.

The top trees resulting from the worst-case algorithm proposed in Chapter 3 have no such constraint. There can be an arbitrary number of *rake* nodes between two consecutive clusters belonging to the same solid path. In Figure 2.6, for example, the base node gi and the *compress* node ei are elements of the same solid path (from e to n), but there are two *rake* nodes between them.

This is an example of a contraction that the worst-case algorithm can represent, but the self-adjusting version cannot. The converse can also happen: some top trees represented by the self-adjusting algorithm would never be created by the worst-case version. For example, the self-adjusting algorithm can, after some sequence of operations, create a top tree with height greater than $\Theta(\log n)$. These two implementations of the top tree interface are not in any way equivalent.

4.6 Final Remarks

This chapter presented a self-adjusting data structure for maintaining dynamic trees. Because it implements the top tree interface, it is as general as any other data structure for this problem. Furthermore, it does so using path decomposition directly, which results in a representation with very little overhead, as a single tree with a direct correspondence (even one-to-one, depending on the implementation) between nodes and clusters.

The bottom-up implementation of top trees described in Chapter 3, in contrast, represents a round-based contraction scheme, which is much stricter. Its implementation requires keeping an Euler tour to represent each level, as well as dummy nodes to represent unmatched clusters. This is similar to what happens with topology trees.

Besides the fact that it makes the data structure conceptually simpler to update, the top-down view offered by path decomposition makes it clear how the data structure can be simplified to handle common special cases. As shown in Section 4.4.1, it is relatively straightforward to specialize self-adjusting top trees to handle applications in which adjacency lists are unordered or in which only path operations need to be performed. It is not obvious how this can be done for contraction-based worst-case top trees.

One could conceivably use path decomposition to implement worst-case top trees by using globally biased binary search trees instead of splay trees to represent *compress* trees. As already mentioned, these data structures are significantly more complicated. Moreover, the fact that splay trees work by bringing accessed nodes to the root greatly simplifies the implementation of dynamic trees. Biased search trees are much less flexible.

Chapter 5

Experimental Analysis

We have seen that top trees are more generic and easier to use than previous data structures, such as ST-trees and ET-trees. This chapter presents experiments whose main goal is to evaluate how much this costs in terms of performance. Both versions of top trees are compared to ST-trees and ET-trees, and also to each other. As one would expect, the simpler amortized version is usually faster than the worst-case one. Perhaps surprisingly, however, there are restricted situations in which the worst-case version is superior.

Another important question is: how practical are dynamic trees in general? All of the data structures presented in Chapter 2 are relatively complicated. The experiments will show that, in practice, it is often worth it to rely on simple linear-time implementations of the dynamic tree operations.

This chapter is organized as follows. Section 5.1 describes the experimental setup. Section 5.2 briefly describes the implementation of the data structures tested. We then present experimental studies of three different applications that use dynamic tree data structures: maximum flows (Section 5.3), online minimum spanning forests (Section 5.4), and single-source shortest paths (Section 5.5). Section 5.6 presents an additional experiment with random dynamic tree operations. Section 5.7 briefly summarizes the experimental results previously reported in the literature and compares them with the ones described here. Final remarks are made in Section 5.8.

5.1 Experimental Setup

All data structures and algorithms were implemented by the author in C++ and compiled with `g++` version 3.4.4 with full optimization (`-O4`). The experiments were performed on a Pentium IV running Gentoo Linux 2.16.14.2 at 2.0 GHz with 1 GB of RAM, 8 KB of L1 cache and 512 KB of L2 cache. While the tests were running, no other CPU-intensive process was being executed.

All times reported are CPU times measured with the `getrusage` function, whose precision is 1/60 of a second. To determine the running time of a given computation, we actually ran it several times until the aggregate time was at least two seconds. To determine the time of each individual run, we divide the aggregate time (which we measured directly) by the number of runs. To ensure all runs were essentially equivalent, the timed executions were preceded by a single untimed run, used to warm up the cache.

We stress that all "runs" mentioned above are calls to a specific function, not calls to the entire program. The program is called only once and has an internal loop that calls the timed functions. The running times do not include the time to generate or read the input data (which is done only once by the entire program), but they do include the time to allocate, initialize, and destroy the data structures (each done once per run within the program).

Input graphs for the maximum flow algorithm studied in Section 5.3 were generated with Anderson's WASHINGTON generator [11]. The remaining inputs were generated by the author. The pseudorandom number generator used was Matsumoto and Nishimura's *Mersenne Twister* [42].

5.2 Data Structures

As already mentioned, the experimental analysis includes top trees, ST-trees, and ET-trees, as well as linear-time implementations of the ST-tree interface. Reasonable effort was made to make them efficient, but further improvements might be possible. When feasible, the data structures share common subroutines. In particular, the routine for splaying on a binary tree was implemented only once as a template function, and used by ST-trees, ET-trees, and top trees. The values maintained by the data structures (such as edge weights and vertex distances) are stored as 64-bit `double`s; switching to 32-bit integers would not have a major effect on performance.

All data structures are initialized with the number of vertices n in the forest they represent; this number does not change during the execution. This allows the data structure to preallocate, when it is created, all the memory it can possibly need. For instance, worst-case top trees need no more than $7n$ clusters, and ET-trees no more than $3n$ nodes. Of course, not all elements (nodes or clusters) are needed at all times. Therefore, each data structure maintains a list of available elements. Whenever a new element is needed, it is removed from this list; when an element is no longer necessary, it returns to the list. The list is implemented as a stack, which means that the most recently discarded element will be the first to be reused.

All data structures were implemented with full functionality; no simplified versions were tested. For example, the ST-tree implementation always supports both *evert* and the ability to add a constant to all vertices of the path, even though the maximum-flow algorithm we tested does not need the former, and the online minimum spanning forest application does not need the latter. Similarly, ET-trees support the ability of finding the minimum-value vertex in a subtree, but the single-source shortest path application (the only one to use ET-trees) does not need it. None of the applications tested requires ordered adjacency lists, but both implementations of top trees support it.

The subsections that follow discuss specific details of each data structure implemented.

5.2.1 ST-trees

Self-adjusting ST-trees were implemented with costs on vertices, following the description in [52].[1] This implementation will be referred to as ST-V. As mentioned in Section 2.1.4, it

[1] Actually, the implementation of *addcost* differs slightly from [52]: a command to update Δmin is missing from the original paper.

can also support costs on arcs as long as no *evert* is ever performed. It suffices to interpret the cost of a vertex v as the cost of the arc from v to its parent, and to make the costs of all root vertices infinite.

The experiments, however, do include an application that requires both arc costs and the ability to change roots (*evert*). For this, we use another version of the data structure, which we call ST-E. Instead of implementing ST-E from scratch, we made it an interface to ST-V. An arc $a = (v, w)$ with cost x in the original tree is represented in the ST-V data structure as a dummy vertex D_a with cost x and exactly two incident arcs, (v, D_a) and (w, D_a). The nodes representing the original vertices of the tree have cost ∞. The ST-E interface merely converts operations in the original forest into operations in the transformed forest, and vice-versa.

Although ST-E is not a direct implementation of ST-trees with costs on edges (such as the one described in [51]), it does have the same size: each vertex or edge of the original forest becomes a node in the data structure. Some of the operations, however, might be slightly costlier. For instance, to *cut* an arc a from the tree, we must *cut* both edges incident to the dummy vertex D_a in ST-V. Similarly, every *link* in the original tree will correspond to two links in ST-V. Because of splaying, the second call is likely to be cheaper than the first, but some overhead is to be expected.

The experiments also include a direct implementation of the ST-tree interface, in which rooted trees are represented explicitly: each vertex maintains a pointer to its parent and an associated cost. With this representation, operations *link*, *cut*, *findcost*, and *parent* can be implemented in constant time. The other operations (*findroot*, *evert*, *findmin*, and *addcost*) require traversing the path from a vertex to the root of its tree, however, and therefore take time linear on the length of this path. To make this implementation consistent with the self-adjusting version of ST-trees, *link*(v, w) also takes linear time, since it must first confirm that *findroot*$(w) \neq v$, i.e., that v and w actually belong to different components.

As in the self-adjusting case, two versions of this linear-time implementation were tested: LIN-V, which associates values with vertices, and LIN-E, which associates values with arcs.

Both actually store values on vertices, and differ only in how these values are interpreted: in LIN-V, the value represents the cost of the vertex itself, whereas in LIN-E it represents the cost of the arc between v and its parent. Both variants implement most operations in essentially the same way, with the exception of *evert*(v). While LIN-V simply reverses the arcs of the path from v to the root by changing the parent pointers of all vertices visited, LIN-E also has to change the costs of these vertices. More precisely, the cost of an arc (v, w), originally stored at v, must be stored at w after the arc is reversed during an *evert*.

5.2.2 ET-trees

The implementation of ET-trees used in the experiment is the self-adjusting version outlined in Section 2.3 (and described in detail in [56]). Each original tree is represented as a splay tree where each original arc appears twice and each original vertex once. In total, no more than $3n$ nodes are used. We refer to this implementation as ET-S.

5.2.3 Top Trees

We implemented both versions of top trees for the experiments. The contraction-based implementation, which supports each operation in $O(\log n)$ worst-case time, will be referred to as TOP-W. It implements *expose* by building temporary trees, as explained in Section 3.3.5. The self-adjusting version is called TOP-S. Both take as input the total number of vertices, a data type (which defines what fields should be maintained in each cluster), and a *processor*, an object that implements methods *create*, *join*, *split*, and *destroy*. Note that, since one of the inputs to the data structures is a type, they must be implemented as templates. One could use virtual functions instead, but, because these cannot be inlined, the performance would be much worse.

The data structure calls the appropriate method from the processor whenever it needs to update the values in some cluster. Because the processor is an object (and not just a collection of functions), it can easily keep a *state*, which is useful in many applications. For example, the single-source shortest path algorithm studied in Section 5.5 needs to maintain a separate array to handle the values of exposed vertices. This array is maintained inside the processor.

For each application tested we developed a different processor, and used a different data type to represent the fields to be stored. Given a particular application, the same processor can be used by both top tree implementations.

Implementing the ST-tree interface. Top trees are generic enough to implement all the functions of the ST-tree interface. In [8], Alstrup et al. detail how this can be done. To maintain rooted trees, they suggest adopting the convention that the second exposed vertex in each component (there are two) always represents its root vertex. To implement the ST-tree operations efficiently, we maintain in each cluster $C = (v, w)$ the following fields:

- $\mathit{mincost}(C)$: the cost of the minimum-cost edge on the path from v to w;
- $\mathit{extra}(C)$: a value to be added to all clusters that represent subpaths of $v \cdots w$;
- $\mathit{extreme}_v(C)$ and $\mathit{extreme}_w(C)$: the base clusters representing the edges incident to v and w, respectively, on the path from v to w;
- $\mathit{minvertex}_v(C)$ and $\mathit{minvertex}_w(C)$: among the two endpoints of the minimum-cost edge on the path from v to w, $\mathit{minvertex}_v$ is the one farthest from v, and $\mathit{minvertex}_w$ the one farthest from w. If there are ties, $\mathit{minvertex}_v$ will refer to the minimum edge that is closer to v, and $\mathit{minvertex}_w$ to the one closer to w.

With these fields, implementing each ST-tree operation is straightforward. We give a few examples. To implement $\mathit{evert}(v)$, one would call $\mathit{expose}(\cdot, v)$. To implement $\mathit{findroot}(v)$, one would call $C \leftarrow \mathit{expose}(v, \cdot)$ and return the second endpoint of C. To implement $\mathit{findmin}(v)$, one would call $C \leftarrow \mathit{expose}(v, \cdot)$ and return $\mathit{minvertex}_w(C)$, where w is the second endpoint of C. For $\mathit{parent}(v)$, one would call $C \leftarrow \mathit{expose}(v, \cdot)$, and return the endpoint of $\mathit{extreme}_v(C)$ that differs from v. The remaining operations can be implemented similarly.

This approach does make it very convenient to use the data structure: the same implementation of the maximum flow algorithm can use either ST-trees or top trees seamlessly. It may have a negative effect on performance, however. The ST-tree interface is natural for how ST-trees are implemented but much less so for top trees. For instance, ST-tree operations are vertex-oriented, with the *parent* operation used to access an edge. Top trees, on the other hand, can manipulate edges directly in a much more natural way. Moreover, the implementation proposed by Alstrup et al. assumes that top trees can remember not only which vertices are exposed in each component, but also which, among the two, is the first exposed vertex and which is the second. Implementing this is relatively straightforward, but it makes the data structure more complicated, and technically this feature is not a requirement of the top tree interface.

In the experiments, we refer to the implementation of the ST-tree interface on top of self-adjusting top trees as TOPST-S, and on top of worst-case top trees as TOPST-W. Both implement the same interface as ST-E. We shall see that they are considerably slower than TOP-S and TOP-W even on the application for which ST-trees were originally designed, a maximum-flow algorithm.

5.3 Maximum Flows

When developing ST-trees, the first data structure to support every dynamic tree operation in $O(\log n)$ time, one of the main motivations of Sleator and Tarjan was to make maximum flow algorithms more efficient. In particular, in [51] they present an algorithm to compute a blocking flow on an acyclic graph in $O(m \log n)$ time, where m is the number of arcs and n the number of vertices in the graph.[2] This can be used as a subroutine of an $O(nm \log n)$-time implementation of Dinic's maximum flow algorithm [19].

The experiments in this section test the dynamic tree data structures on a different maximum flow algorithm, based on *distance labels* (which we shall describe shortly). Al-

[2]A blocking flow is a flow from the source s to the sink t of an acyclic graph graph such that each s-t path contains an edge whose flow equals its capacity.

though it does not use the notion of blocking flows explicitly, the algorithm is very similar to Dinic's algorithm, as mentioned in [5, Section 7.5]. It uses dynamic trees in a similar way and achieves the same worst-case running time. It is, however, simpler to implement, as Section 5.3.2 will show.

5.3.1 Basics

To describe the algorithm, let us first formalize the problem.[3] Let $G = (V, A)$ be a directed graph with two distinguished vertices, a *source* s and a *sink* t (reachable from s). Each arc (i, j) has an associated positive *capacity* c_{ij}. To simplify notation, assume that $c_{ij} = 0$ for every pair (i, j) that is not an arc. A *flow* f is an assignment $A \mapsto \mathcal{R}$ such that, for all i and j, $0 \leq f_{ij} \leq c_{ij}$ and, for every vertex i (except s and t), $\sum_j f_{ji} = \sum_k f_{ik}$. The *value* of the flow is $\sum_i f_{si}$, which is equal to $\sum_i f_{it}$. The *maximum flow problem* is that of determining a flow of maximum value.

Given any flow f, we define the *residual capacity* of an arc (i, j) as $r_{ij} = c_{ij} - f_{ij}$. Intuitively, the residual capacity represents how much more flow the arc supports. When the residual capacity is zero, the arc is said to be *saturated*. For most maximum flow algorithms, it helps to assume that, for every original arc (i, j) with flow f_{ij} there is also a reverse arc (j, i) with residual capacity f_{ij}. Therefore, reducing the flow on (i, j) can be interpreted as increasing the flow on (j, i). The *residual network* of G is the subgraph of G induced by the arcs (both original and reverse) with strictly positive residual capacity.

The residual capacity of a path in G is defined as the minimum residual capacity of its arcs. If a path from s to t has positive residual capacity, we say that it is an *augmenting path*. Based on this notion, one can define a simple algorithm for the maximum flow problem: In each step, find an augmenting path and send as much flow as possible along it; repeat until no augmenting path exists. It is not hard to prove that this generic algorithm will eventually find the maximum flow, but the algorithm will not be strongly polynomial if the choice of augmenting paths is left unconstrained. Fortunately, there are slightly more

[3]We assume the reader is familiar with the basic concepts of network flows, so the description will be very terse. For a more complete discussion, see [5], for example.

sophisticated algorithms that can guarantee polynomial-time convergence, such as the one described in the next section.

5.3.2 The Shortest Augmenting Path Algorithm

The algorithm implemented for the experiments is the *shortest augmenting path* algorithm, due to Edmonds and Karp [21]. In each iteration, it performs an augmentation along the s-t path in the residual network that has the fewest arcs (ties are broken arbitrarily). To find this path efficiently, we use the notion of *distance labels*. The distance label of a vertex v, denoted by $d(v)$, is a lower bound on the distance in the residual network (in number of arcs) from v to the sink t. An arc (i, j) is called *admissible* if $d(i) = d(j)+1$. The *admissible network* is the subgraph of the residual network containing only admissible arcs. It is easy to see that any path from s to t in the admissible network is a shortest augmenting path.

The algorithm works as follows. It starts at the source s and grows a path one vertex at a time, always picking an admissible outgoing arc. Whenever it reaches a dead end (i.e., a vertex with no outgoing admissible arc), the algorithm backtracks and tries a different outgoing arc from the previous vertex. Eventually, it will either reach t (and perform an augmentation) or confirm that no augmenting path exists.

More precisely, let v be the current vertex being processed by the algorithm. Initially, $v \leftarrow s$. At the beginning of an iteration, we check whether there is an admissible arc (v, w) leaving v. If there is, we *advance* by setting $v \leftarrow w$ and $pred(w) \leftarrow v$, where *pred* denotes the predecessor on the current tentative path; if $v = t$ after advancing, we augment and start over from s. If there is no outgoing admissible arc from v, we *retreat*: first we update $d(v)$ (i.e., we set $d(v) \leftarrow d(u) + 1$, where u is, among all vertices for which (v, u) is not saturated, the one with the minimum distance label), then set $v \leftarrow pred(v)$.

The algorithm proceeds until $d(s) = n$, at which point the admissible network is guaranteed to have no augmenting path. A tighter stopping criterion in practice is the *gap heuristic*: stop when there is an integer i between 0 and $d(s)$ such that there are no vertices with distance label i. It is easy to implement this stopping criterion using counters to keep track of the number of vertices with each distance label at any time.

The running time of the maximum flow algorithm can be shown to be $O(m^2 n)$. If one maintains a *current arc data structure*, which keeps track of the most recent arc taken out of each vertex, the total running time can be reduced to $O(n^2 m)$. The reader is referred to [5] for further details on the algorithm and its analysis.

Our implementation uses both the gap heuristic and the current arc data structure.

Using ST-Trees

One of the "n" factors in the $O(n^2 m)$ expression accounts for the fact that each augmentation may take $\Theta(n)$ time, since an augmenting path may have up to $\Theta(n)$ arcs. If we use a dynamic tree data structure, the time per augmentation is reduced to $O(\log n)$, even for such long paths.

The data structure is used to maintain a forest of admissible arcs. Each tree of the forest is rooted, and every arc (i, j) is such that j is the parent of i in the tree and $d(i) = d(j) + 1$. The cost of (i, j) in the forest represents its residual capacity in the graph. In every iteration, the current vertex v will be the root of the tree that contains s. Initially, $v \leftarrow s$ and the forest has no arcs.

Each iteration starts by verifying if there is an admissible arc (v, w) leaving v. If there is, we *advance* by adding this arc to the forest (with the *link* operation) with cost r_{vw} and making $v \leftarrow$ *findroot*(w). If there is no admissible arc leaving v, we *retreat* by removing from the forest (with *cut*) all arcs incident to v, updating the distance label of v (we set $d(v) \leftarrow d(u) + 1$, where u is, among all vertices for which (v, u) is not saturated, the one with the minimum distance label), and, finally, setting $v \leftarrow$ *findroot*(s). To execute this step efficiently, it is convenient to maintain for each vertex v a list of all incoming arcs in the forest.

After advancing, we check whether the current vertex v is t. If it is, we have found an augmenting path from s to t and must send as much flow as possible along it. We do this with dynamic tree operations: we call $v \leftarrow$ *findmin*(s) and $\Delta \leftarrow$ *findcost*(v) to determine the amount Δ by which the flow can be increased (i.e., the residual capacity of the path), then *addcost*$(s, -\Delta)$ to update the residual capacity of all arcs on the path. We then remove all newly saturated arcs from the forest by *cutting* all arcs that have zero residual capacity. These arcs can be found with repeated calls to *findmin*(s).

Note that the version of dynamic trees required for this application has very restricted properties: the data structure must maintain rooted trees, and the roots do not change during the algorithm (except due to *link* and *cut*); there is no *evert* operation. Furthermore, data must be aggregated only along paths, not trees. Not coincidentally, these are the properties that are handled naturally by ST-trees.

An important aspect of this application is that structural operations (*link* and *cut*) constitute a large fraction of all operations. Every non-structural operation (*findmin*, *addcost*, *findcost*, or *findroot*) is soon followed by a structural operation.

Using Top Trees

The obvious way of using top trees within the shortest augmenting path maximum flow algorithm is to implement the ST-tree interface on top of it. As already mentioned, this may have an adverse effect on performance. A better alternative is to define the internal operations of top trees so as to satisfy the specific needs of the maximum flow application, but without implementing the ST-tree interface directly.

There is more than one way of doing this; we describe the one used in the experiments. Since the application has to represent directed trees with no *evert* operation, one can interpret each cluster C of the top tree as representing both a *directed* path and a *rooted* subtree of the original tree. More precisely, the application maintains the following fields in each cluster:

- *root*(C): This is the root vertex of the subtree represented by cluster C.
- *minarc*(C): If *root*(C) is an endpoint of C, *minarc*(C) is a pointer to the base cluster representing the minimum-cost arc on the path from the other endpoint of C to the

root. If there is a tie, the arc closest to the root is picked. If $\mathit{root}(C)$ is not an endpoint of C, $\mathit{minarc}(C)$ is *null* (undefined).

- $\mathit{mincost}(C)$: If $\mathit{minarc}(C)$ is not *null*, $\mathit{mincost}(C)$ is the cost (i.e., residual capacity) of the minimum arc. Otherwise, $\mathit{mincost}(C)$ is undefined.

- $\mathit{extra}(C)$: A lazy value to be added to all arcs on the path represented by C. It will always be zero if $\mathit{minarc}(C)$ is undefined.

In general, all arcs in the subtree represented by C will be directed towards one of the endpoints of C (its root). The only exception is if the cluster actually contains, as an internal vertex, the root r of the entire tree. In this case, $\mathit{root}(C)$ will be r and both $\mathit{minarc}(C)$ and $\mathit{mincost}(C)$ will be undefined. To ensure that these invariants hold, the internal operations are defined as follows:

- $C \leftarrow \mathit{create}(e)$: Let the endpoints of e be v and w, and assume the edge is directed from v to w. The base cluster C will be initialized with $\mathit{root}(C) \leftarrow w$, $\mathit{minarc}(C) \leftarrow C$, $\mathit{mincost}(C) \leftarrow r_{vw}$, and $\mathit{extra}(C) \leftarrow 0$.

- $C \leftarrow \mathit{join}(A, B)$. Let $A = (u, v)$ and $B = (v, w)$. We always set $\mathit{extra}(C) \leftarrow 0$, but the other fields depend on whether $\mathit{minarc}(A)$ and $\mathit{minarc}(B)$ are both defined.

  If one of them is not defined, the other will be: *minarc* is only undefined for clusters that contain the root as an internal vertex, and A and B intersect only at v. We set $\mathit{root}(C)$ to be the root of the cluster (either A or B) whose *minarc* field is undefined, and keep both $\mathit{minarc}(C)$ and $\mathit{mincost}(C)$ undefined.

  If both $\mathit{minarc}(A)$ and $\mathit{minarc}(B)$ are defined, the outcome depends on the move being performed:

  1. The move is a *rake*. This means that u will disappear. If $\mathit{root}(A) = u$, we set $\mathit{root}(C) \leftarrow u$ and keep $\mathit{minarc}(C)$ and $\mathit{mincost}(C)$ undefined. Otherwise, we set $\mathit{root}(C)$, $\mathit{minarc}(C)$, and $\mathit{mincost}(C)$ to the corresponding values in B.

  2. The move is a *compress*. There are two subcases to consider.

(a) $\mathit{root}(A) \neq \mathit{root}(B)$. In this case, paths (u, v) and (v, w) are oriented consistently, i.e., either $u \rightarrow v \rightarrow w$ or $w \rightarrow v \rightarrow u$. These cases are symmetric, so assume the first orientation holds. We set $\mathit{root}(C) \leftarrow w$ and make $\mathit{minarc}(C)$ and $\mathit{mincost}(C)$ refer to the minimum among the two clusters. If there is a tie, we set $\mathit{minarc}(C) \leftarrow \mathit{minarc}(B)$, since B is closer to the root than A is.

(b) $\mathit{root}(A) = \mathit{root}(B)$. In this case, v must be the root of the entire tree, and the two paths are not oriented consistently: they both converge to v. We set $\mathit{root}(C) \leftarrow v$, but keep both $\mathit{minarc}(C)$ and $\mathit{mincost}(C)$ undefined.

- $(A, B) \leftarrow \mathit{split}(C)$. If C is a *compress* cluster, we add $\mathit{extra}(C)$ to the *extra* and *mincost* fields of both child clusters. If C is a *rake* cluster, we add $\mathit{extra}(C)$ to $\mathit{extra}(B)$ and $\mathit{mincost}(B)$ only. All other fields remain unchanged.

- $\mathit{destroy}(C)$ does nothing.

With these fields, using top trees within the shortest augmenting path network flow algorithm is straightforward. For example, to find the root of the subtree containing some vertex v (which is necessary after an advance or retreat), it suffices to call $C \leftarrow \mathit{expose}(v, \cdot)$ and pick $\mathit{root}(C)$.

To perform an augmentation, we first call $C \leftarrow \mathit{expose}(s, t)$. The current residual capacity of the path is $c \leftarrow \mathit{mincost}(C)$. We decrement both $\mathit{extra}(C)$ and $\mathit{mincost}(C)$ by c. We must then remove every arc that has zero residual capacity from the path represented by C. While $\mathit{mincost}(C) = 0$, we do the following: (1) let $M \leftarrow \mathit{minarc}(C)$; (2) call $\mathit{cut}(M)$; and (3) set $C \leftarrow \mathit{expose}(s, u)$, where u is the endpoint of M that is farthest from the root (i.e., the tail of the directed arc M represents). At the end of this loop, we return to the main loop of the algorithm from vertex $v \leftarrow \mathit{root}(C)$.

Note that a single call to *expose* is enough to retrieve the cost of the minimum-capacity arc on a path, the arc itself, and the root of the path. ST-trees require calls to separate functions (*findmin*, *findcost*, and *findroot*) to achieve the same goal. This may be a potential advantage of top trees over ST-trees in this application. Of course, this is an interface issue only. The top tree interface dictates that as much information as possible should be made available at the root. In contrast, each ST-tree operation gathers a single piece of information, but one could conceivably change the interface of ST-trees to allow it to gather more information in each access. In some sense, that is what self-adjusting top trees do.

5.3.3 Experimental Results

The first class of graphs on which the maximum flow algorithm was tested is that of *random layered graphs*, created with Anderson's WASHINGTON generator [11]. These graphs are defined by two structural parameters, the number of rows (r) and the number of columns (c). From each vertex in column i ($1 \leq i < c$) there are outgoing arcs to three vertices picked at random in column $i+1$. The source s is connected by an outgoing arc to all vertices in the first column, and the sink t has an incoming arc from each vertex in the last column. Arcs incident to s and t have infinite capacity; all others have integral capacities chosen uniformly at random from the interval $[1; 2^{20}]$. Note that the graph is acyclic, with $n = rc + 2$ and $m = 3r(c-1) + 2r = 3rc - r$.

In our experiments, we used $r = 4$ and varied c from 128 to 32 768. The fact that r is a small constant ensures that every augmenting path will have $\Theta(n)$ vertices. For each set of parameters, we generated five graphs with different random seeds. We ran each algorithm on all five graphs and computed the average time. The results are shown in Table 5.1. Figure 5.1 refers to the same experiment, with times given as multiples of ST-V.

Among the $O(\log n)$-time dynamic tree implementations, vertex-based self-adjusting ST-trees (ST-V) led to the fastest algorithm on all cases. Even though the application associates values with edges, we can use this data structure here because the *evert* operation is not needed. This is fortunate, since it is almost twice as fast as the more generic edge-based self-adjusting ST-trees (ST-E). In fact, ST-E is even slower than self-adjusting top trees (TOP-S) when the number of vertices is small; only when the graph is big enough does ST-E, which is more cache-efficient, become (slightly) faster.

Among application-specific top trees, the self-adjusting variant (TOP-S) is 3 to 4 times

Table 5.1: Performance of the shortest augmenting path maximum flow algorithm (implemented with different data structures) on random layered graphs with four rows. Times are averages taken over five graphs and given in milliseconds. The best time for each input size is marked in bold.

VERTICES	LIN-V	ST-V	ST-E	TOP-S	TOPST-S	TOP-W	TOPST-W
514	**1.2**	5.3	10.9	8.7	22.9	35.3	71.3
1026	**4.8**	11.2	23.2	19.4	48.3	82.7	160.3
2050	**16.1**	20.6	42.1	39.0	92.0	160.9	343.6
4098	62.3	**42.8**	85.4	89.6	193.0	341.2	787.3
8194	230.4	**95.2**	186.8	206.3	407.5	818.3	1693.6
16386	903.4	**189.9**	351.9	439.5	805.4	1481.6	3466.1
32770	6588.8	**440.3**	808.6	1041.6	1804.6	3714.2	7714.4
65538	59247.2	**806.5**	1444.3	1972.5	3312.3	6369.6	15302.5
131074	229333.1	**1610.4**	2862.6	3919.4	6323.8	13964.3	30632.5

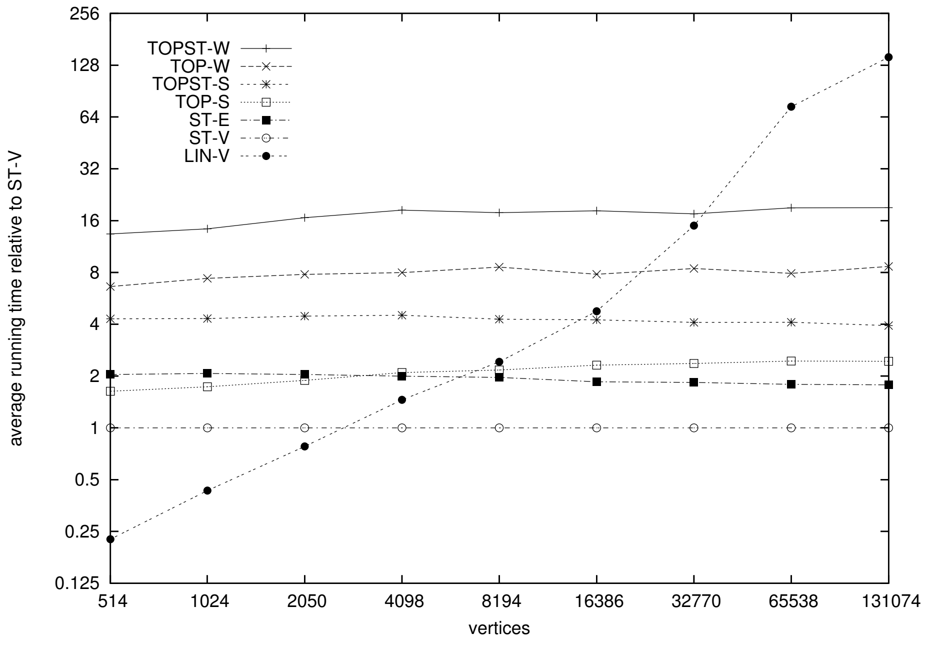


Figure 5.1: Running times of the maximum flow algorithm on random layered graphs with four rows; all times are relative to ST-V.

faster than the worst-case implementation (TOP-W), which is not surprising: the latter is a more involved algorithm. Moreover, in this application consecutive calls to the dynamic tree data structure are heavily correlated: the algorithm performs a series of operations involving a single path in the tree until it becomes an augmenting path. Because of splaying, self-adjusting data structures are better equipped to take advantage of this locality of access.

As anticipated, application-specific top trees are significantly more efficient than top trees with the ST-tree interface. Comparing the worst-case versions, we see TOPST-W is more than twice as slow as TOP-W. The difference between the self-adjusting versions (TOPST-S and TOP-S) is slightly smaller, especially when the number of vertices is large. This discrepancy can once again be explained by the fact that the extra calls made to implement the ST-tree interface are correlated, and self-adjusting data structures can take more advantage of this. Even though TOPST-S performs more operations than TOP-S, these operations are still well correlated.

Finally, we observe that the obvious, linear-time implementation of the ST-tree interface (LIN-V) does rather well on small graphs: up to around 3000 vertices, it leads to the fastest algorithm. Since the length of the augmenting paths grows linearly with graph size, however, the algorithm is eventually surpassed by the other methods.

To test the maximum flow algorithms on graphs with smaller diameter, we used directed square meshes, also created by the WASHINGTON generator. A mesh with parameter k has a special source s, a special sink t, and k^2 other vertices arranged as a square grid. Each of these k^2 vertices has an outgoing arc to each of its up to four neighbors in the grid. Vertices on the border will have fewer grid neighbors. In addition, there is an outgoing arc from s to each vertex of the first column of the grid and an outgoing arc from each vertex of the last column to t. The arcs incident to s and t have infinite capacity; all other capacities are integers picked uniformly at random from the interval $[1; 2^{20}]$. Nine values of k were tested (16, 22, 32, 45, 64, 90, 128, 181, and 256) with five graphs each (with different random seeds). Since graph size is proportional to k^2, it roughly doubles from one value of k to the next.

Table 5.2: Performance of the shortest augmenting path maximum flow algorithm on square meshes (with additional source and sink). All times given in milliseconds. The best time for each input size is marked in bold.

VERTICES	LIN-V	ST-V	ST-E	TOP-S	TOPST-S	TOP-W	TOPST-W
258	**0.7**	3.8	8.5	5.6	16.0	19.6	30.3
486	**1.7**	8.4	18.3	12.6	34.6	48.6	72.3
1026	**4.4**	20.0	45.2	31.2	82.6	135.8	192.0
2027	**13.6**	52.2	115.2	82.2	206.1	373.3	521.4
4098	**55.3**	167.8	356.7	272.2	605.0	1247.4	1660.9
8102	**218.4**	506.8	1041.5	781.5	1616.4	3836.8	4703.5
16386	**992.3**	1835.4	3567.9	2668.6	5216.2	12964.8	16030.6
32763	**3495.3**	5333.4	9939.3	7543.7	13926.3	38102.0	47082.0
65538	**16040.0**	23617.4	43374.4	30434.8	59309.8	176539.8	207077.3

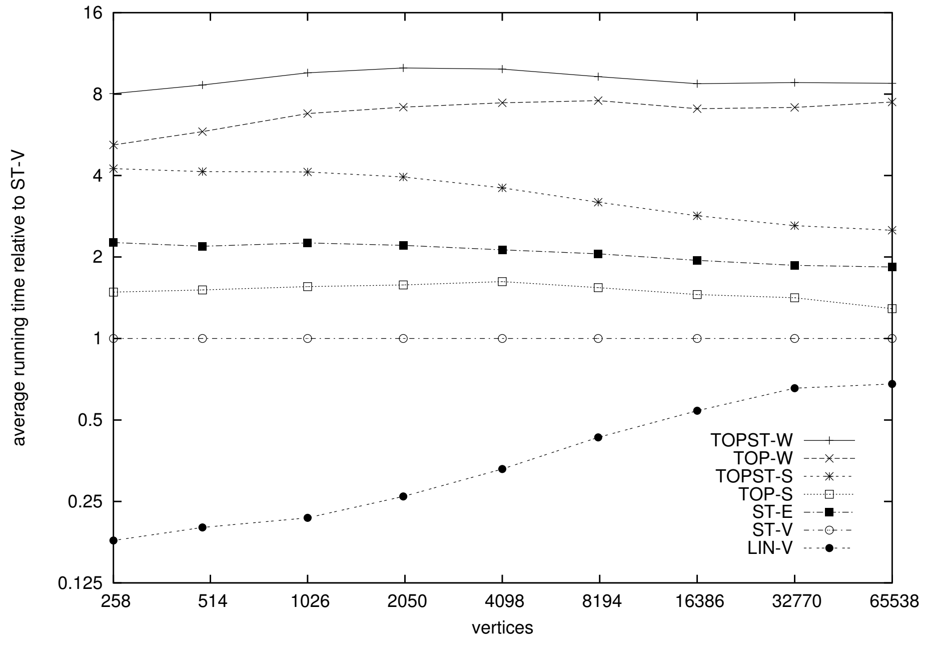


Figure 5.2: Running times of the maximum flow algorithm on square meshes; all times are relative to ST-V.

Table 5.2 shows the average running times of the maximum flow algorithm with various dynamic-tree data structures. Figure 5.2 refers to the same experiment, but with running times given with respect to ST-V.

The results are consistent with those obtained for random layered graphs, with one main exception: now that augmenting paths have length $\Omega(\sqrt{n})$, LIN-V remains the fastest alternative for much larger graphs. Its speedup with respect to the $O(\log n)$ data structures does get smaller as n increases, but with as many as 65 538 vertices LIN-V is still the best option.

Also, in this class of graphs TOP-S is consistently faster than ST-E, and for larger graphs it takes less than 50% more time than ST-V. This good performance can be explained by the fact that the top tree interface allows us to gather more information from the root cluster. A single call to *expose* gives us access to the root vertex, the minimum arc, and the cost of the minimum arc on the path. In contrast, with ST-V one must call three separate methods to get this information.

5.4 Online Minimum Spanning Forests

The second application we consider is the *online minimum spanning forest* problem, also known as the *semi-dynamic spanning forest* problem. The goal is to maintain the minimum spanning forest of a graph to which edges are added one at a time. The number of vertices in the graph (n) is given in advance, but the set of edges is initially empty. We denote the total number of edges eventually added to the graph by m.

5.4.1 The Algorithm

Dynamic tree data structures can be used to process each new edge in $O(\log n)$ time. At all times, the data structure maintains the current minimum spanning forest (MSF). When a new edge $e = (v, w)$ with cost c is added to the graph, we determine if v and w are in the same component of the MSF. If they are not, we just add e to the forest. If they are in the same component, we determine the maximum-cost edge e' on the current path between v

and w in the MSF. If e' costs more than c, we remove e' from the forest and add e; otherwise we discard e. This algorithm is a straightforward application of the "red rule" described by Tarjan in [54, Chapter 6]. It states that, if an edge is the most expensive of *some* cycle in the graph, then it does not belong to the minimum spanning forest.

Implementation with ST-trees. To determine if v and w are in the same component using ST-trees, we first call $\mathit{evert}(v)$ then check if $\mathit{findroot}(w) = v$. If this is not true, we simply call $\mathit{link}(v, w, c)$. Otherwise, we call $u \leftarrow \mathit{findmax}(w)$, then $c' \leftarrow \mathit{findcost}(u)$. If $c' > c$, we perform $\mathit{cut}(u)$, then $\mathit{link}(v, w, c)$.

Observe that *findmax* technically does not belong to the ST-tree interface. Although implementing it would be relatively easy (it is analogous to *findmin*), we can actually preserve the original interface by negating all the weights when dealing with the data structure. Whenever an edge with cost c is to be inserted into the tree, we do it with cost $-c$. Whenever we need to call *findmax*, we call *findmin*. Finally, if *findcost* returns a value c, we interpret it as $-c$.

Since the algorithm needs the *evert* operation, we cannot use the simplified version of ST-trees that associates values with vertices (ST-V). We must use ST-E, which explicitly maintains values on edges.

Implementation with top trees. To implement the algorithm using top trees, it suffices to keep two pieces of information in each cluster $C = (v, w)$: the cost of the most expensive edge on the path from v to w, and a pointer to the base cluster representing this edge. Ties are broken arbitrarily. Implementing the internal top tree operations to maintain these values appropriately is trivial: the only actual processing occurs when the *join* operation processes a *compress* cluster, when it copies the information from the child with maximum cost.

To process an edge (v, w) with cost c, the online minimum spanning forest algorithm first calls $C \leftarrow \mathit{expose}(v, w)$. If $C = \mathit{null}$ (meaning that v and w are in different components) we *link* v and w. Otherwise, if the value stored in C is greater than c, we *cut* the maximum-length edge and use *link* to add (v, w) to the forest.

5.4.2 Experimental Setup

Four different data structures were tested in the solution of the online minimum spanning tree problem: edge-based self-adjusting ST-trees (ST-E), the obvious linear time implementation of the ST-tree interface (LIN-E), worst-case top trees (TOP-W), and self-adjusting top trees (TOP-S). Both versions of top trees are application-specific: TOPST-S and TOPST-W, which are much slower, have not been tested.

For reference, we also ran Kruskal's algorithm for the (offline) minimum spanning tree problem [18, 40, 54]. Our implementation copies the entire list of edges to a temporary array, sorts them by cost with quicksort with the median-of-three pivot-selection strategy [18, 49], then processes them in this order. Every edge that does not create a cycle is marked as belonging to the forest. The algorithm stops when $n-1$ edges are inserted or (if the graph is disconnected) when all m edges are processed. To keep track of connected components and detect cycles, we use a union-find data structure with path compression and union by rank [18, 54]. It ensures that each edge is processed in $\alpha(m, n)$ amortized time, where α is an extremely slow-growing functional inverse of Ackermann's function. The running time of the algorithm is therefore dominated by the time to sort the input list, which is $O(m \log m)$. The online algorithms have the same asymptotic bound (as long as m is a polynomial function of n, which is the case in our experiments), but we shall see that the constants associated with Kruskal's algorithm are much lower.

In the maximum flow algorithm studied in Section 5.3, the number of queries to the forest was roughly the same as the number of structural operations (*links* and *cuts*). With online minimum spanning forests, this is not necessarily true, since a query (finding the maximum-weight edge on a path) might not be followed by a *link* or *cut*. As a result, queries may be asymptotically more numerous than structural operations. In fact, we will see that the number of structural operations performed by the algorithm when processing a fixed graph depends on the order in which the edges are processed.

5.4.3 Random Graphs

To create a random graph with n vertices and m edges, we generate each edge in turn by picking a pair of distinct endpoints uniformly at random. Note that the graph is not necessarily connected and that multiple edges between the same pair of vertices are allowed. Edge costs are also picked uniformly at random from the interval $[1; 1000]$. For every set of parameters, five graphs with different random seeds were tested; we report the average results.

In our first experiment, we fixed the number of vertices at $n = 4096$ and varied m, the number of edges. Figure 5.3 shows how the performance of the MSF algorithm depends on m when edges are processed in random order. For each value of m, the plot shows the average time to process an edge.

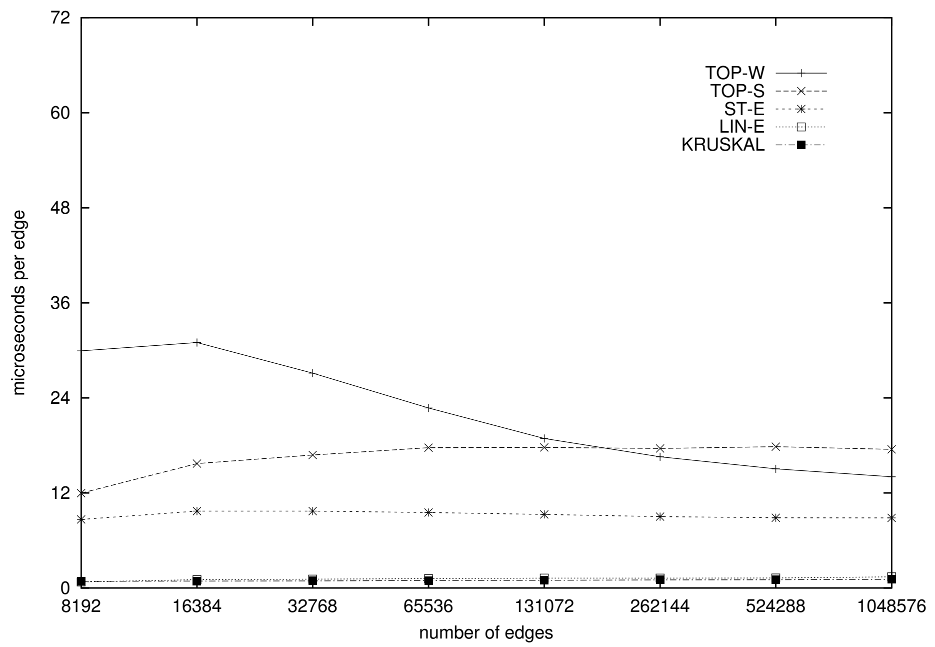


Figure 5.3: Online minimum spanning forests: Average processing time per edge for random graphs with 4096 vertices. Edges are processed in random order.

The first observation to make is that LIN-E is significantly faster than all $O(\log n)$-time data structures. Because the graphs are random, all paths traversed are expected to have only $O(\log n)$ edges. The running time of Kruskal's algorithm is comparable to that of LIN-E; it is even slightly faster when the number of edges is large.

When there are few edges, the relative performance of the other data structures is the same as with maximum-flow algorithms: ST-E is faster than TOP-S, which is faster than TOP-W. As the number of edges increases, however, TOP-W eventually becomes faster than TOP-S. The reason is that the fraction of edges that are actually inserted into the forest decreases as the graph becomes denser. More precisely, if m is the number of edges processed, the expected fraction of edges that are actually inserted is $O((\log m)/m)$:

Lemma 13 *Let G be a random weighted multigraph on n vertices and m edges, with edge weights assigned independently at random. If the edges are processed in random order, the expected number of edges actually inserted by the online minimum spanning forest algorithm (i.e., the number of* links*) is $O(n \log m)$.*

Proof. Let e_i be the i-th edge processed by the algorithm, and let p_i be the probability that it is inserted into the current minimum spanning forest. By definition, this is exactly the probability that e_i belongs to the minimum spanning forest of the subgraph G_i of G containing only the first i edges of the sequence. Because the sequence is random, G_i is also a random multigraph with random edge weights. Every edge in the sequence has equal probability of belonging to the minimum spanning forest. Since the forest has no more than $n-1$ edges, $p_i < n/i$. By linearity of expectation, the total number of insertions is bounded by $\sum_{i=1}^{m} p_i < \sum_{i=1}^{m} n/i = O(n \log m)$. □

When m is large enough, the running time of the online algorithms is dominated by *expose* operations, which are much cheaper for the worst-case data structure than *links* and *cuts* are. Recall that *expose* does not require TOP-W to actually rebuild the contraction: it just builds a temporary tree on the side.

Figure 5.4 refers to the same experiment: random graphs with 4096 vertices and edges processed in random order. Instead of comparing running times, it shows the average number of calls to *join* per edge as a function of the total number of edges processed.

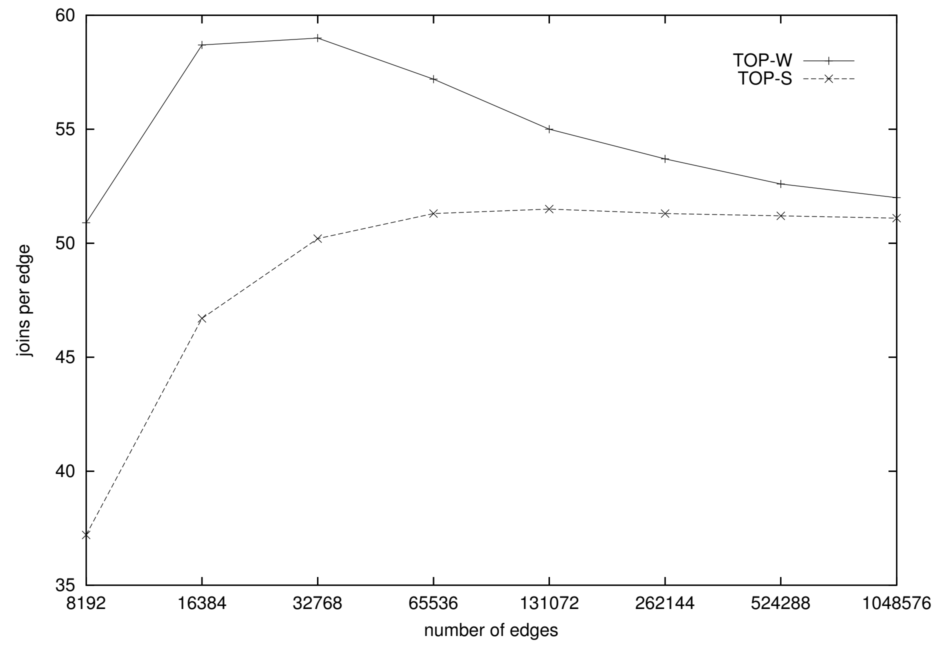


Figure 5.4: Online minimum spanning forests: Average number of calls to *join* per edge processed on random graphs with 4096 vertices.

The plot shows that TOP-W must create more new clusters to process each edge in the input than TOP-S does. As the number of edges increases, the difference between the algorithms starts to decrease. The average *expose* operation in TOP-W clearly affects fewer clusters than in TOP-S, but *links* and *cuts* in TOP-W are so much more expensive that TOP-S still dominates with more than one million edges. Comparing Figures 5.3 and 5.4, we see that not only does TOP-W need to manipulate more clusters than TOP-S when performing structural operations, but the running time per cluster is also larger. When $m = 16\,384$, for instance, TOP-W executes only 30% more *joins* on average, but is twice as slow. The situations is reversed when *expose* operations are more numerous: with 1 048 576 edges, TOP-W still executes more operations on average, but it is faster.

In another experiment with the same graphs, we consider what happens when edges are processed in increasing order of cost. In this case, there will be no more than $n - 1$ *links*, and *cut* will never be called. One expects an even greater fraction of the running time to be dominated by *expose* operations. Figure 5.5 confirms that the performance of TOP-W is even better than when edges are processed in random order. The self-adjusting algorithms (ST-E and TOP-S) are also faster than before, but by a much smaller factor. Both LIN-E and Kruskal's algorithm remain largely unaffected.

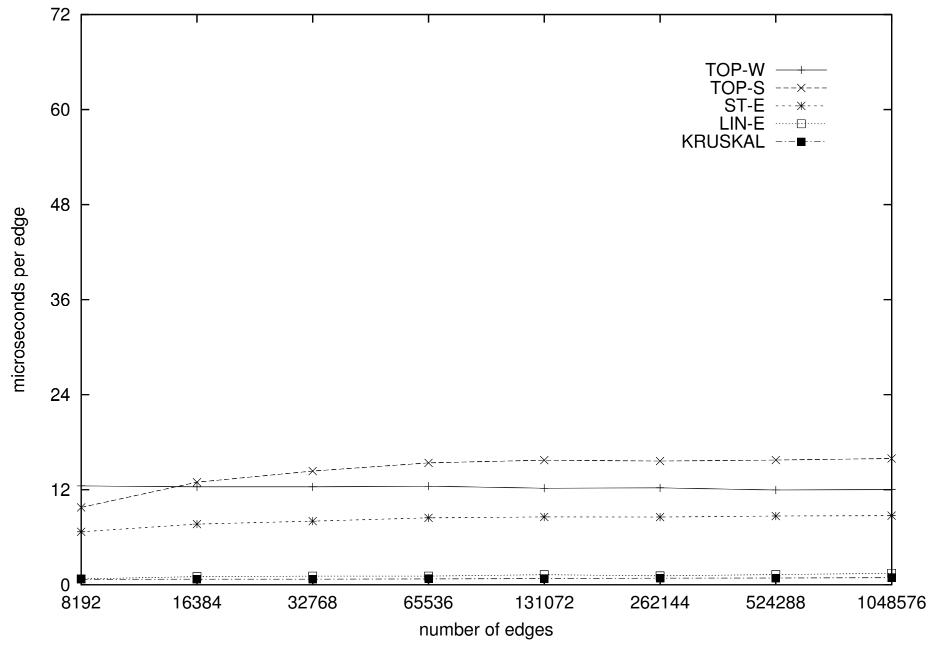


Figure 5.5: Online minimum spanning forests: Average processing time per edge for random graphs with 4096 vertices. Edges are processed in increasing order of cost.

Figure 5.6 presents the opposite extreme. When edges are processed in decreasing order of cost, every edge will cause a structural operation, except in the rare case of a tie. All algorithms are significantly affected, but none more than TOP-W, which becomes almost five times slower than when edges are given in increasing order. Not only do the self-adjusting data structures have simpler update algorithms (they do not need to maintain Euler tours, for example), but they also benefit from the fact that structural operations are always performed on paths that are already exposed. Due to splaying, the relevant nodes tend to be closer to the root of the top tree.

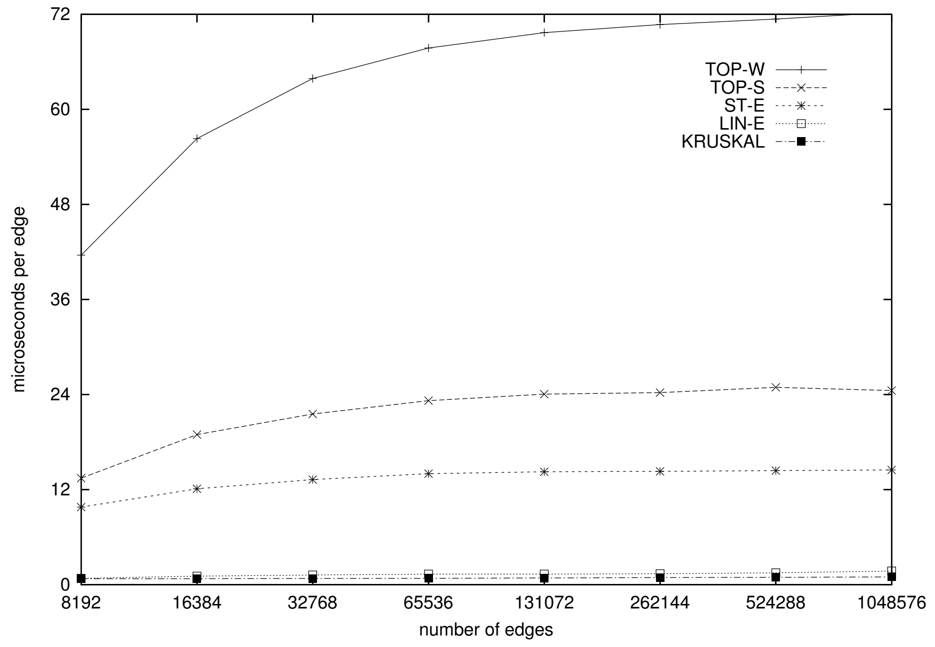


Figure 5.6: Online minimum spanning forests: Average processing time per edge for random graphs with 4096 vertices. Edges are processed in decreasing order of cost.

5.4.4 Circular Meshes

We also tested the minimum spanning forest algorithm on circular meshes. In a circular mesh with r rows and c columns, the vertices are organized as an $r \times c$ grid, with edges between each vertex and its four neighbors. To handle the borders, we consider the first and last columns to be adjacent; the same holds for the first and last rows. The number of vertices is $n = rc$ and the number of edges is $m = 2rc$. Edge weights are picked at random from the interval $[1; 1000]$. In the experiment, edges were always processed in random order. Once again, we generated five graphs for each set of parameters and report the average results.

We first consider symmetric circular meshes, i.e., those with $r = c$. Figure 5.7 shows the average time necessary to process an edge when r and c vary from 16 to 512.

An important difference between the results for meshes and random graphs is that on meshes the performance of TOP-W is relatively worse, compared to the other methods. Because the graph is very sparse, *links* and *cuts* make up a much larger fraction of the

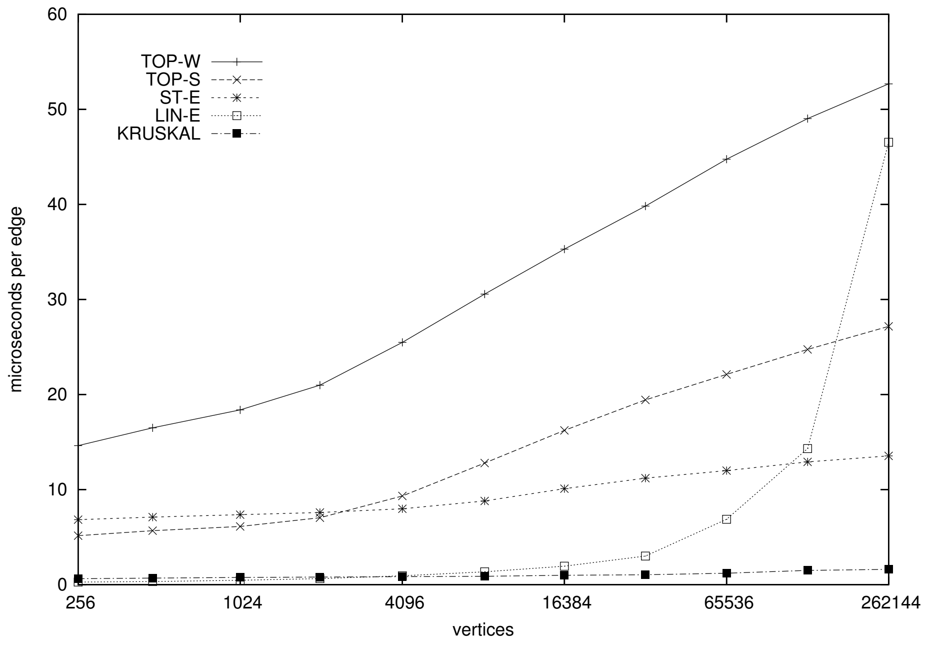


Figure 5.7: Online minimum spanning forests: Average processing time per edge for symmetric circular meshes with varying number of vertices.

operations than on denser random graphs.

But the most obvious difference between these results and those obtained for random graphs is that the linear-time implementation of ST-trees (LIN-E) is no longer superior to the other data structures for every value of n. This can be explained by the fact that symmetric circular meshes have diameter $\Theta(\sqrt{n})$, as opposed to $\Theta(\log n)$ for random graphs. Once n is large enough, the linear-time algorithm ceases to be the best option. This only happens for values of n as high as 100 000, however.

It should be noted that the *findmin* operation is not the bottleneck of LIN-E. When an edge (v, w) is examined, the expected length of the path between v and w is actually quite small, given the restricted topology of meshes. In the experiment, the average length of the exposed paths increases very slowly, from 7.5 edges when $n = 256$ to 46.2 edges when $n = 262\,144$. However, before actually calling *findmin*(v), we must call *evert*(w), which reverses the path between w and the current root. Since the current root is a random vertex (one of the endpoints of the previous edge processed), the expected length of this path is proportional to the diameter of the tree. In the experiments, the average number of vertices visited by *evert* ranged from 13.1 to 790.3, thus making *evert* the bottleneck of the algorithm.

To better understand how the algorithms depend on the diameter of the graph, we performed a second experiment in which the number of vertices n was kept constant at 65 536, but the number of rows and columns varied. We tested values from $r = 2$ (and $c = 32\,768$) to $r = 256$ (and $c = 256$). The results are presented in Figure 5.8.

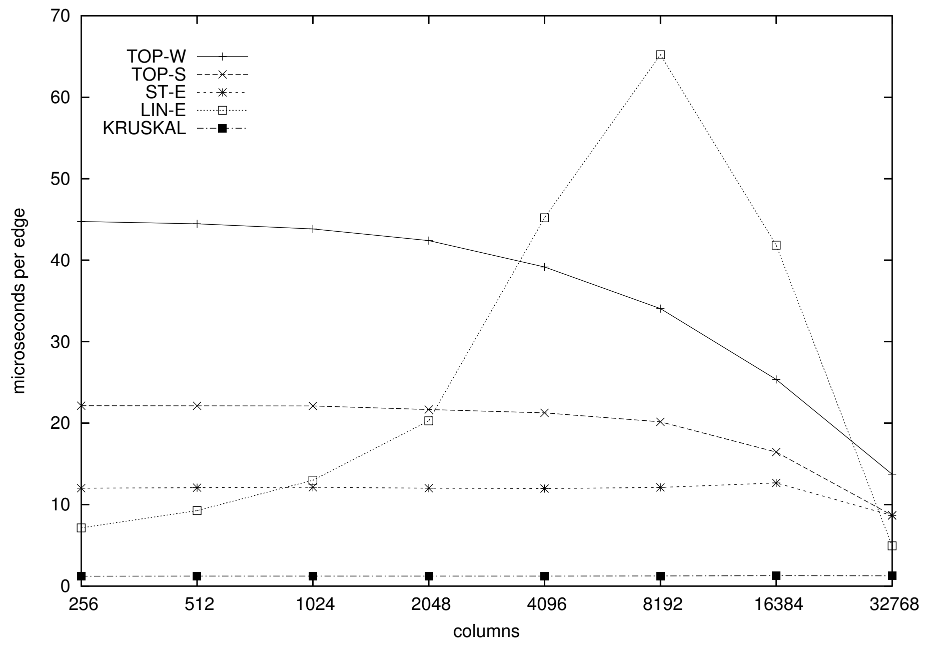


Figure 5.8: Online minimum spanning forests: Average processing time per edge for circular meshes with 65 536 vertices and varying number c of columns. The number of rows, which is $65536/c$, varies accordingly.

Once again, the behavior of LIN-E stands out. The diameter of the graph is proportional to the number of columns. An increase in this number should hurt the performance of the algorithm, and this is indeed what happens as long as the number of columns is not too large. When it is large, there are very few rows, and as a result the graph has $\Omega(n)$ balanced cuts of very small size (i.e., cuts with few edges that separate the graph into two large components). Since the edges are processed in random order, it takes longer until very large components are formed. The algorithm will eventually have to deal with components that have very high diameter, but it spends most of its time processing much smaller components. The fact that it takes longer for large components to form also explains why the other implementations (using ST-E, TOP-S, and TOP-W) become faster as the number of rows decreases. As expected, changes in the graph topology have no measurable effect on the performance of Kruskal's algorithm.

5.4.5 High-Degree Vertices

An important difference between self-adjusting top trees and ST-trees is how they handle vertices of high degree. While ST-trees use dashed edges to link children directly to their high-degree parent, self-adjusting top trees must build *rake* trees to aggregate information stored in the children.

To assess how high-degree vertices affect the relative performance of the data structures, we tested the online minimum spanning forest algorithm on *augmented stars*. An augmented star is a graph with three parameters: the number of *spokes* (which we denote by k), the length of each spoke (ℓ), and the total number of edges (m, which must be at least $k\ell$). It contains a central vertex from which k paths of length ℓ (the spokes) emanate; we call this subgraph a *star*. The remaining $m - k\ell$ edges consist of pairs of distinct vertices picked at random from the spokes. Parallel edges are allowed. Figure 5.9 shows an augmented star with $k = 8$ spokes of length $\ell = 6$.

In the experiments, the costs of the $k\ell$ edges on the star were set to 1 and the costs of the remaining edges to 2. In order to create a vertex of high degree as soon as possible, we make the minimum spanning forest algorithm process the edges of the star first, and only then, in random order, the remaining edges. Only the first $n-1$ edges will be inserted into the tree. The others will essentially amount to queries: to test an edge (v, w), the data structure must find the most expensive edge on the path between v and w in the star. For large values of k, this path is very likely to contain the center, which at this point will have

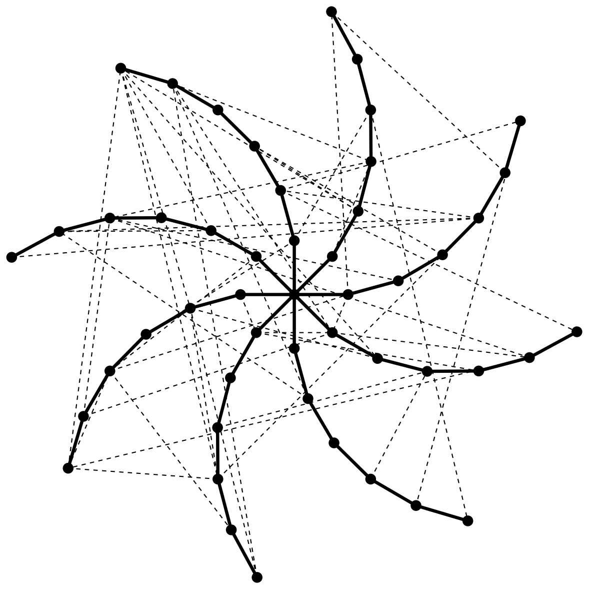


Figure 5.9: An extended star with eight spokes of six vertices each. The star itself is the minimum spanning tree, and its edges (represented by solid lines) are the first to be processed in the experiment. The additional edges (represented by dashed lines) link random pairs of vertices and are never added to the minimum spanning tree.

very high degree within the tree.

We first consider graphs of fixed size ($n = 65537$ and $m = 655370$), but with a varying number k of spokes. Of course, the length ℓ of each spoke must vary as well: for any fixed value of k, ℓ must be $65536/k$. Figure 5.10 shows the average time to process an edge as a function of k. Five graphs were tested for each value of k.

For $k = 2$, the star has only two spokes, which makes it a path on n vertices. As k increases, the diameter of the tree decreases, becoming as small as 2 when $k = 65536$. This has an obvious effect on the linear-time data structure, whose running time per operation is proportional to the diameter. Eventually, LIN-E becomes three times as fast as Kruskal's algorithm. ST-trees also benefit from an increase in k. When $k = \Theta(n)$, there will be $\Theta(n)$ splay trees of constant size linked by dashed edges to the node representing the central vertex of the star. When $k = 65536$, the algorithm is four times as fast as when $k = 2$.

In contrast, top trees do not benefit from an increase in k. As already mentioned, self-adjusting top trees must maintain a *rake* tree (or two, because of the circular order) to

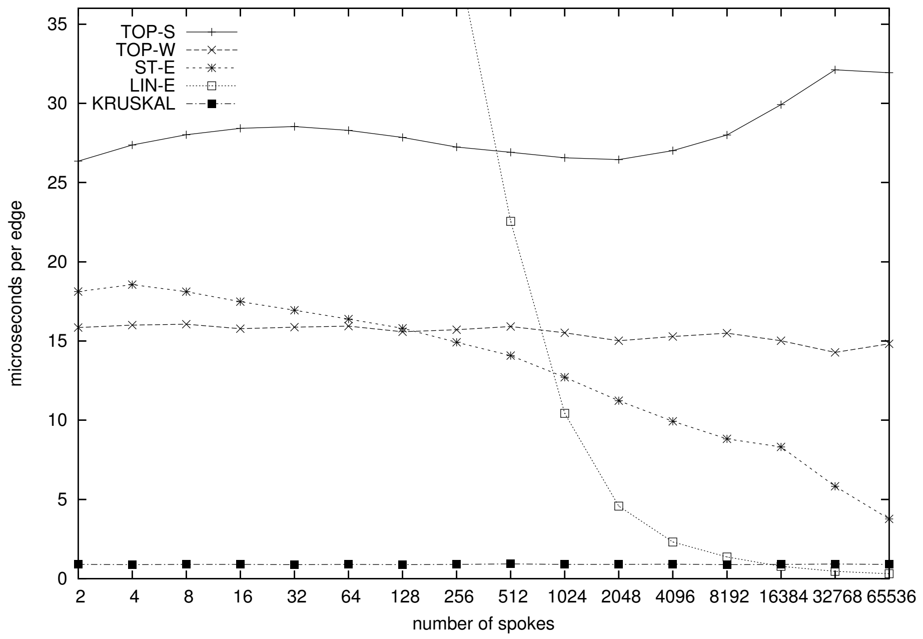


Figure 5.10: Online minimum spanning forests: Average processing time per edge for augmented stars with 65537 vertices and varying number of spokes.

aggregate the information about the edges incident to the high-degree vertex, and accessing this tree already takes logarithmic time. Worst-case top trees do not have explicit *rake* trees, but they still have logarithmic height regardless of whether more *rakes* or *compresses* are performed. Once again, the performance of Kruskal's algorithm does not depend on the topology of the graph.

With the same class of graphs, we performed a second experiment in which the length ℓ of each spoke was fixed at 64, but the number of spokes varied from $k = 2$ to $k = 1024$. The total number of vertices varied accordingly, from 128 to 65 536. In each case, the number of edges in the graph was set to $10n$. Figure 5.11 shows how the average time to process an edge varies as k increases.

In this experiment, one would expect the time it takes for ST-E to process an edge to be independent of the number of spokes. Indeed, this is what happens up to around 32 spokes, at which point the running time slowly starts to increase. As Section 5.4.6 will show, this can be attributed to cache effects: queries on larger trees have less locality, and therefore

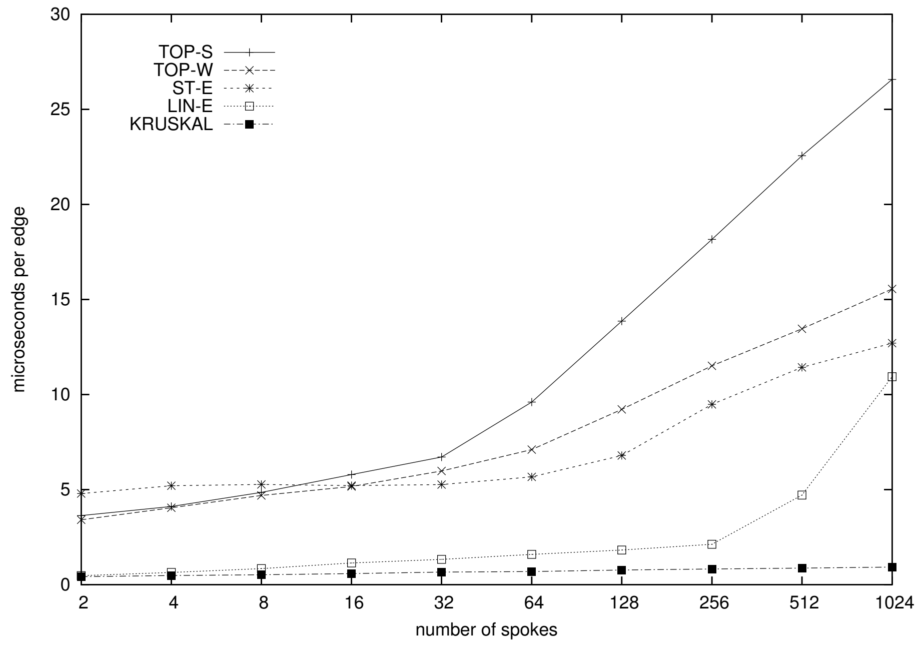


Figure 5.11: Online minimum spanning forests: Average processing time per edge for augmented stars with spokes of length 64.

access main memory more often.

Cache effects also explain why the slopes of the curves representing TOP-W, TOP-S, and LIN-E increase when the number of spokes reaches a certain value. Note that the inflection point is approximately 32 for TOP-W, 64 for TOP-S, and 256 for LIN-E: the more space-efficient the algorithm is, the later the inflection point is reached.

5.4.6 Memory Usage and Cache Effects

In this section, we investigate in more detail how the performance of the data structures is affected by the memory hierarchy. If there were no memory hierarchy (i.e., no cache), running the online minimum spanning tree algorithm on a random graph on n nodes and m edges should be no more expensive (per edge) than processing two or more copies of this graph at once. Due to caching, one should expect all algorithms to slow down as the number of copies increases.

To test this hypothesis, we created a family of graphs with three parameters: the number of components (c), the number of vertices per component (n'), and the average number of edges per component (m'). The graph will have $n = cn'$ vertices, partitioned (at random) into c components with n' vertices each, and $m = cm'$ edges in total. Each edge is generated by first picking a component uniformly at random, then a pair of vertices from this component (also at random). Edge costs are drawn from the interval $[1; 1000]$. Figure 5.12 shows the running time (per edge) of the online minimum spanning forest algorithm using $n' = 32$ and $m' = 128$, with the number of components varying from 1 to 65 536.

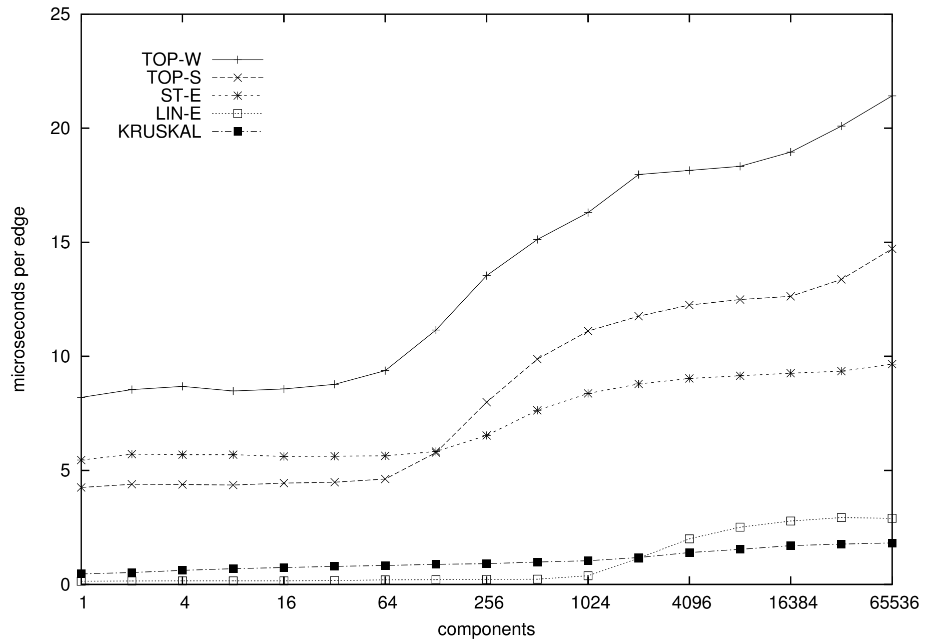


Figure 5.12: Online minimum spanning forests: Performance of the online minimum spanning forest algorithm on graphs consisting of 32-vertex components with 128 edges each (on average). The number of components varies.

In theory, the average running time per edge of all online algorithms should be independent of c. As long as the number of components is small, this is indeed what happens. However, once c is large enough, every algorithm starts to get slower as c increases—this indicates that the data structures no longer fit in cache, and main memory is accessed more often. Eventually, all curves essentially level off, indicating that most of the computation happens in main memory. The running time of some of the algorithms (notably TOP-W and TOP-S) actually starts increasing again after this plateau is reached, however, which indicates that they occasionally make use of virtual memory.

The plot makes it clear that data structures that require more bytes per vertex are affected sooner: TOP-W reaches an inflection point first, then TOP-S, then ST-E, and finally LIN-E. Interestingly, the data structure whose performance is most dependent on the memory hierarchy is the simplest one, LIN-E. When the number of components is small, LIN-E can process each edge in around 0.2 microseconds—the actual value is 0.23 for $c = 512$. Due to cache effects, however, the algorithm becomes almost 15 times slower: with 16 384 components, each edge is processed in 2.9 microseconds on average. The other data structures only slow down by a factor of 2 to 3.

The LIN-E algorithm is more dependent on cache precisely because it is so simple: it essentially does nothing *but* memory accesses. Most of its methods consist of following parent pointers, with little (if any) additional computation. If its nodes do not fit in cache, the algorithm will spend most of its time accessing main memory. The other data structures must access more data, but included in their original running time is the time to actually process the data: performing rotations, testing for new moves, updating vertex ID's, and so on. When the number of vertices increases, the time to process the data does not increase as much as the time to fetch it from memory. The overall effect on performance is therefore not as pronounced as with LIN-E.

Kruskal's algorithm has running time $O(c \log c)$ when n' and m' are constant, which is asymptotically worse than the online algorithms. Because it has better locality, however, it actually becomes the fastest of all algorithms (even LIN-E) when operating in main memory, at least for the graph sizes tested.

5.5 Single-Source Shortest Paths

Both applications discussed so far, maximum flows and online minimum spanning trees, require dynamic trees to perform only path operations. We now consider an application that requires information to be aggregated over entire trees: a label-correcting single-source shortest path algorithm. Given a directed graph $G = (V, A)$ (with $|V| = n$ and $|A| = m$) with arbitrary arc lengths $\ell(\cdot)$ and a source $s \in V$, the algorithm either finds the distances from s to every vertex in V, or detects a negative cycle if there is one. A negative cycle is a directed cycle such that the sum of the lengths of its arcs is negative.

5.5.1 Algorithm

A generic label-correcting algorithm assigns to each vertex v a *distance label* $d(v)$ representing an upper bound on the distance from s to v.[4] Initially, we set $d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for all other vertices v. When an arc $(v, w) \in A$ is such that $d(w) > d(v) + \ell(v, w)$, we can *relax* it by setting $d(w) \leftarrow d(v) + \ell(v, w)$. While there are arcs that can be relaxed, the algorithm picks one and relaxes it. If there are no negative cycles, eventually there will be no arc to be relaxed, and the algorithm terminates with *exact* distance labels, i.e., $d(v)$ will represent the actual distance from s, for all v.

The algorithm will be correct regardless of which arc is relaxed in each step. To guarantee that it will be efficient, however, we need a more restrictive selection scheme. A possible rule is to arrange the arcs in some fixed order (it does not matter which) and to work in *passes*. Each pass processes each arc in the list exactly once, relaxing those that can be relaxed. The algorithm stops as soon as there is a pass that relaxes no arc. This rule ensures that, after k passes, all shortest paths with k arcs or fewer will have been found. When there are no negative cycles, no shortest path will have more than $n - 1$ arcs, which means the algorithm will never execute more than n passes. In fact, an edge will be relaxed during the n-th pass if and only if there is a negative cycle in the graph. Since each pass can be implemented in $O(m)$ time, the total running time of this algorithm (due to Bellman [12]) is $O(mn)$.

When we relax an arc (v, w), we are effectively making v the parent of w in a candidate shortest path tree rooted at s. Moreover, we are reducing the distance label of w by some

[4]Once again, the description of the algorithm will be terse; for more details, see [5], for example.

value Δ. At this point, we know that the distance labels of all descendants of w in the current tree could also be reduced by at least Δ. This is not done automatically, however; it may take several passes until all the arcs in the subtree are processed in the correct order and the distance labels are updated.

Using ET-trees. The updates can be performed automatically with a dynamic tree data structure that allows aggregation over trees. This means that standard ST-trees cannot be used, but ET-trees can. The data structure is used to represent the current tentative shortest path tree, and the value of a vertex refers to its distance label. To process an arc (v, w), we call *findval*(v) and *findval*(w) to find the distance labels of v and w. If $d(w) < d(v) + \ell(v, w)$, we relax the arc as follows: (1) *cut* the arc between w and its current parent; (2) call *addval*$(w, d(v) + \ell(v, w) - d(w))$ (thus decreasing the distance labels of all edges in the subtree rooted at w); and (3) *link* the new edge. The data structure interface is defined so that the *link* operation fails if (v, w) creates a cycle; since this can only happen when the cycle is negative, we can stop at this point.

Unfortunately, this algorithm does not reduce the number of passes that may be necessary in the worst case: it is still $\Theta(n)$. Since each edge can be processed in $O(\log n)$ time using ET-trees, the total running time of the algorithm is $O(mn \log n)$, which is actually worse than Bellman's original implementation. The algorithm using dynamic trees could conceivably require fewer passes to converge, however. Moreover, it detects a negative cycle as soon as one is created, thus eliminating the need to keep the algorithm running until the n-th pass.

In practice, a simple modification of Bellman's algorithm—periodically running a linear-time cycle-detecting routine—will also eliminate the need to run all n passes and will be much faster than the algorithm using dynamic trees. For an experimental evaluation of this and other algorithms for this problem, the reader is referred to [15].

Using top trees. Evidently, we can use top trees instead of ET-trees. We keep a global array $a[\cdot]$ of size n to represent distance labels and maintain a single field *extra*(C) in each cluster C representing a value to be added to all internal vertices of the cluster. When *create* or *join* are applied to the cluster, this value is set to zero. When it is *split*, we increment the *extra* fields of both children by *extra*(C) and do the same for $a[v]$, where v is the vertex eliminated by the move C represents. To query the actual distance label of v, we must first expose v, then look at $a[v]$. To decrement by Δ the distance label of every vertex in a tree rooted at v, we first run $C \leftarrow$ *expose*(v) then decrement *extra*(C), $a[v]$, and $a[w]$ by Δ, where w is the second endpoint of C (besides v).

5.5.2 Experiments

We tested the single-source shortest path algorithm on random graphs with an added Hamiltonian cycle. A graph with n vertices and m edges (with $m \geq n$) is generated by first picking a random circular permutation of the vertices and creating a cycle in which adjacent vertices in the permutation are connected; the remaining $m-n$ edges are then added at random. We always used $m = 4n$ and varied n. For each value of n, five different graphs were tested. Arc lengths are chosen at random from the interval [1;10] for arcs in the original Hamiltonian cycle, and from $[1; 1000]$ for the remaining arcs.

Since all arc lengths are positive, the graph is guaranteed to have no negative cycle. But, as described, it has no negative path either, which makes it less interesting. To create negative paths, we assigned a *potential* $\pi(v)$ chosen at random between -1000 and 1000 to each vertex v in the graph and replaced the length $\ell(v, w)$ of each arc (v, w) by its reduced cost, defined as $\ell'(v, w) = \ell(v, w) - \pi(v) + \pi(w)$. This transformation does not change the length of any cycle in the graph, since the sum of the potentials telescopes. In particular, no negative cycle will be created, but paths with negative lengths will.

Figure 5.13 shows the average time per edge per iteration required by each algorithm. Three of the algorithms (TOP-W, TOP-S, and ET-S) use a dynamic tree data structure; the fourth, which we refer to as BELLMAN, is a direct implementation of Bellman's algorithm. All four algorithms process the edges in the same order in each iteration, but the order is determined at random.

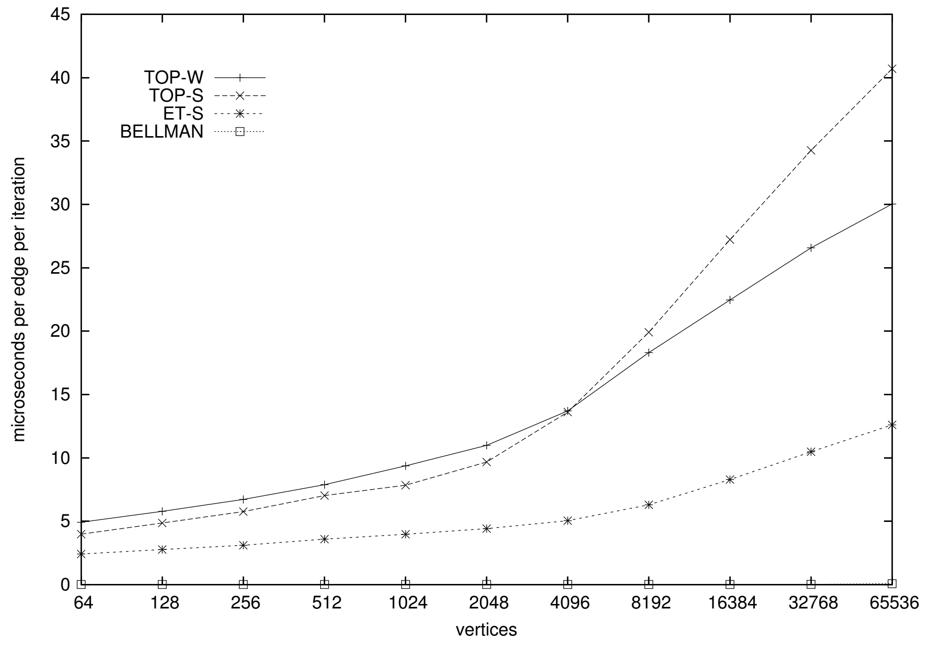


Figure 5.13: Single Source Shortest Paths: Average time to process an edge on random graphs with an added Hamiltonian cycle.

The figure shows that worst-case top trees are competitive with self-adjusting top trees. This indicates that a sizeable portion of the arcs processed are not actually inserted into the tree: only *expose* is performed. Table 5.3 confirms that this is indeed what happens, especially as the number of vertices increases: eventually, less than a tenth of the arcs processed cause an insertion into the tree.

The figure also shows that ET-trees are 2 to 3 times faster than self-adjusting top trees. This can be explained by the fact that ET-trees are more space-efficient and simpler to update. Rather than being a collection of *rake* and *compress* trees, an ET-tree is just a splay tree. A typical query will require a single splay in the tree.

Even though ET-trees are more efficient than the other $O(\log n)$ data structures, finding shortest paths using them is still much slower than using the standard implementation of Bellman's algorithm, which can process an edge hundreds of times faster. As Table 5.3 shows, ET-trees do reduce the number of iterations, but by an amount not nearly enough to offset the extra cost per iteration. For every graph size tested, Bellman's algorithm was

Table 5.3: Single-source shortest paths: For each graph size, the table shows the average number of iterations, the average percentage of arcs that cause *links* to be performed (i.e., are relaxed), and the average running time of the entire algorithm (in milliseconds). The corresponding values for TOP-W and TOP-S are identical to those obtained by ET-S, except for the running time.

	ITERATIONS		LINKS (%)		TOTAL TIME (ms)	
VERTICES	BELLMAN	ET-S	BELLMAN	ET-S	BELLMAN	ET-S
64	15.6	6.0	10.1	12.8	0.04	3.70
128	20.0	6.8	9.0	11.6	0.10	9.63
256	21.0	8.0	9.1	10.3	0.22	25.34
512	26.0	8.6	7.8	9.5	0.52	63.11
1024	28.6	8.8	7.8	10.0	1.21	143.17
2048	31.0	10.0	7.8	9.3	2.68	361.80
4096	35.8	10.2	7.4	9.0	6.40	843.94
8192	39.2	11.4	7.1	8.4	17.04	2351.04
16384	40.8	12.2	7.2	8.2	35.22	6623.59
32768	48.6	12.8	6.5	8.2	119.68	17571.73
65536	49.0	13.4	6.8	7.8	907.20	44273.27

at least 48 times faster than ET-trees.

5.6 Random Operations

The experiments described so far use dynamic trees within more involved algorithms, in which structural operations (*links* and *cuts*) are typically correlated. In the maximum flow application, a query is always followed by a *link* or *cut* involving an edge on the same path. For online minimum spanning forests, a *cut* is always performed on an edge from the path queried immediately before.

In this section, we compare the algorithms when executing a random sequence of *links* and *cuts* on an n-vertex forest, with no queries. A sequence of m operations is determined (*a priori*) as follows. The first $n-1$ operations are *links* that create a random spanning tree. The $m-n+1$ remaining operations are alternating *cuts* and *links*: we remove a random edge from the current tree and replace it with a random edge between the two resulting components.

We repeated this experiment for several values of n, always with $m = 10n$. Since there are no queries, we ran this algorithm with ET-trees, all variants of ST-trees, and top trees. For top trees, we maintained in each cluster the same fields as in the minimum spanning forest application. For each value of n, five input sequences were tested. The average running times per operation are reported in Figure 5.14.

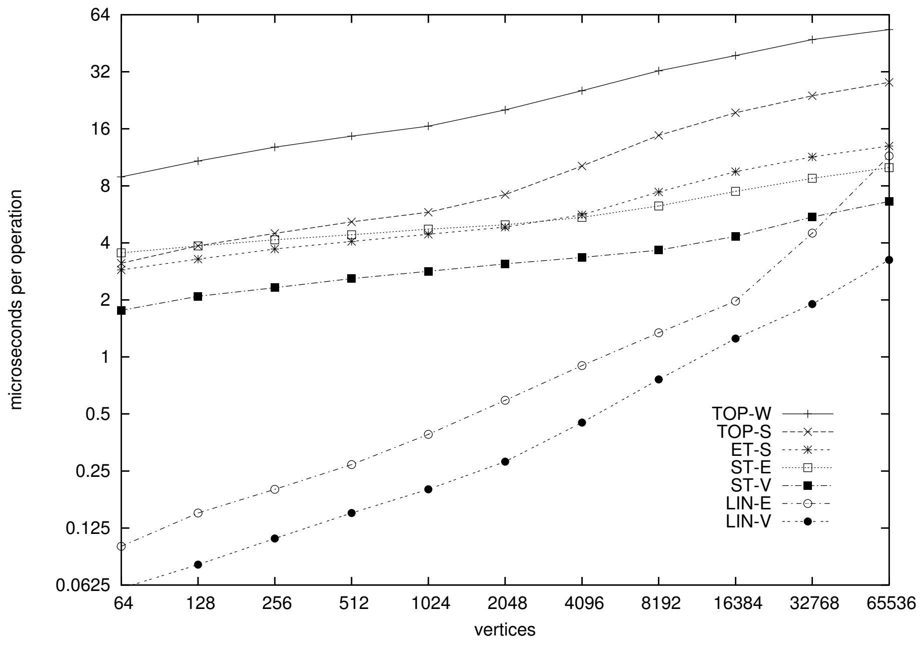


Figure 5.14: Performance of various data structures while executing a randomized sequence of *links* and *cuts*, with no queries.

As one would expect from previous experiments, worst-case top trees (TOP-W) are the slowest among the $O(\log n)$-time data structures. However, TOP-W is only roughly twice as slow as TOP-S; within the maximum flow algorithms, TOP-S is 3 to 4 times faster, as Figures 5.1 and 5.2 show. This confirms that self-adjusting top trees do benefit from the fact that consecutive operations within the maximum flow algorithm tend to be correlated.

Among the other $O(\log n)$-time algorithms, ST-V is the fastest, roughly twice as fast as TOP-S, ST-E, and ET-S when the number of vertices is small. These three algorithms have similar performance when n is small, but when the graph size increases ST-E becomes the fastest among them, followed by ET-S and TOP-S. This is exactly the order one would expect taking just space usage into account.

Generating a random sequence of *links* and *cuts*. An interesting aside to this experiment is how one can generate the test sequence of *links* and *cuts* efficiently. Given a tree with $n - 1$ edges, it is trivial to pick one edge at random to be *cut*. One can, for instance, keep a separate array with a list of the current edges in the tree, and pick an element from this list at random. Suppose the edge is (v, w). We must now select an edge to replace it, which requires picking a vertex at random from each of the two components formed after (v, w) is removed. This can be done trivially in linear time, but one would like to do it more efficiently, in $O(\log n)$ time.

We can do it using top trees. To pick a random vertex from a top tree, it suffices to maintain two fields in each cluster C. The first is $\mathit{intcount}(C)$, the total number of internal vertices in the cluster. The second is $\mathit{rand}(C)$, a random internal vertex. When $\mathit{intcount}(C)$ is zero, $\mathit{rand}(C)$ is undefined. Maintaining $\mathit{intcount}(C)$ is trivial: the *create* operation sets it to zero, and *join* sets it to the sum of the values in the children plus one (to account for the one vertex that becomes internal). Updating $\mathit{rand}(C)$ during a *join* is also straightforward. Let A and B be the child clusters of C, and let v be the vertex that disappears when A and B are combined. We pick $\mathit{rand}(C)$ from the set $\{\mathit{rand}(A), \mathit{rand}(B), v\}$, with probabilities weighted by $\mathit{intcount}(A)$, $\mathit{intcount}(B)$, and 1, respectively.

To pick a vertex at random from the component containing v, we call $C \leftarrow \mathit{expose}(v)$ and pick either $\mathit{rand}(C)$ (with probability weighted by $\mathit{intcount}(C)$) or one of the endpoints of C (each with weight 1). If $r = \mathit{rand}(C)$ is selected, we must call $\mathit{expose}(r)$ after the operation. If we do not do this, the distribution will not be uniform: further attempts to pick random vertices from the same components would tend to return some vertices more often than others.

To see why this is true, it is convenient to interpret the top tree as a tournament between the internal vertices, with $r = \mathit{rand}(R)$ (where R is the root cluster) as the winner. To be the winner, it must have been selected every time a cluster containing it as an internal vertex was joined with another cluster. The purpose of the tournament is to give the application access to a random vertex. Once the application actually picks this vertex, in principle the data structure should simulate another tournament to select another random winner.

We claim this is actually not necessary: it suffices to reevaluate the clusters that have r as the winner, which are exactly the clusters that have r as an internal vertex. The purpose of $\mathit{expose}(r)$ is to *split* these clusters.

Note that these are the only clusters whose randomness we have used. More precisely, let C be a cluster with children A and B. We know that $\mathit{rand}(C)$ may be either $\mathit{rand}(A)$ or $\mathit{rand}(B)$ (or neither one of them). The "choice" between them does not depend on the actual values of the *rand* field; it depends only on the *intcount* fields. Suppose $\mathit{rand}(C) \neq \mathit{rand}(A)$. Then, even though we know what $\mathit{rand}(A)$ is, we do not use it to determine the winner of the "match" between A and B (or of the entire tournament). When a new tournament is needed, the original outcome of the "subtournament" represented by A is still a random variable. Therefore, A does not need to be *split*.

5.7 Previous Work

This section summarizes the experimental results previously reported in the literature.

Topology trees. In [25], Frederickson compares topology trees to a self-adjusting implementation of ST-trees (both "coded by an undergraduate under careful supervision"). The data structures are used within the preflow-push maximum flow algorithm of Goldberg and Tarjan [29]. Topology trees took on average 44% more time than ST-trees to execute the dynamic tree operations. The actual difference ranged from 32% to 60% on graphs with 10 000 nodes and number of arcs ranging from 10 000 to 90 000.

Top trees. In an unpublished manuscript [8], Alstrup, Holm and Thorup perform a preliminary experimental comparison between top trees and other dynamic tree data structures. Unfortunately, although some of the theoretical results presented in this manuscript appeared later in [10], the experimental part has never been published.

Rather than compare the algorithms within an actual application, the authors chose to focus on the data structure itself. Their test procedure consists of building a free tree with n vertices, then going through n *rounds* of operations. Each round consists of picking two vertices v and w at random, performing $\mathit{evert}(w)$ and then calling $\mathit{parent}(v)$, $\mathit{findroot}(v)$, $\mathit{findcost}(v)$, $\mathit{findmin}(v)$, and $\mathit{cut}(v)$. Finally, *link* is called to restore a single component. The input trees were guaranteed to have large diameter (long paths). Three classes of inputs were tested, with diameters $\Theta(\log n)$, $\Theta(\sqrt{n}\log n)$ and $\Theta(n)$. All data structures were implemented with the ST-tree interface with weights on edges.

The authors concluded that self-adjusting ST-trees were the fastest of the data structures tested. They were around 6 times faster than an implementation of ST-trees using weighted treaps [50], which are a randomized version of globally biased search trees that are much simpler to implement but guarantee only expected bounds. Self-adjusting ST-trees were roughly 15 times faster than a direct implementation of top trees (presumably using the update algorithm suggested in [34]).[5] Somewhat surprisingly, topology trees were even slower than top trees, by a factor of approximately 2.5.

They also compared these data structures with the obvious linear-time implementation of the ST-interface. This simple algorithm was the fastest for trees with diameter $\Theta(\log n)$ (by a factor of 5 to 10), regardless of graph size. For graphs with larger diameter, self-adjusting ST-trees are faster for long enough paths (in the order of hundreds of edges). This is in line with the experiments reported here.

RC-trees. In [4], Acar et al. present an experimental evaluation of their implementation of RC-trees, which they compare with this author's implementation of self-adjusting top trees (ST-E).

Their simplest test consisted of building a tree on n nodes and performing a series of *cuts*, each immediately followed by a *link* that restores the original edge. In this case ST-trees are around five times faster than RC-trees, except when almost all vertices in the tree have degree two: in this case, RC-trees become up to 12 times slower than ST-trees. This is

[5]As already mentioned, the authors later found the analysis of the algorithm to be flawed.

due to the randomized nature of *compresses* mentioned in Section 2.3: the probability of a degree-two vertex being eliminated in any given round is only 1/8, regardless of the degree of its neighbors.

If only non-structural operations are performed, RC-trees and ST-trees have similar performance. RC-trees are faster for path queries and when changing edge weights, while ST-trees are faster when adding weights to paths. The data structures are always within a factor of two of one another, however.

The authors also studied more realistic uses of dynamic trees, within two basic applications: online minimum spanning forests and Dinic's maximum flow algorithm (implemented following the description provided by Sleator and Tarjan in [51]).

The maximum flow algorithm was tested on random layered graphs with 5 rows, $n/5$ columns, and capacity of up to 2^{20}. They tested values of n between 400 and 102 400. Self-adjusting ST-trees consistently outperform RC-trees by a factor of eight.

The online minimum spanning forest algorithm was tested on random graphs with 32 768 vertices. The graphs are initially empty, and new edges are processed one at a time (with random endpoints and random weights). After 32 768 edges are processed, ST-trees are more than twice as fast as RC-trees. The difference in performance becomes gradually smaller, until, after about one million edges, RC-trees become faster. The denser the original graph becomes, the less likely it is that a new edge will be inserted, which means that the running time will be dominated by queries. As pointed out by the authors, queries are cheap in RC-trees, since they do not require any changes to the tree. The authors tested graphs with up to 16 million edges, and in the limit RC-trees are about twice as fast as ST-trees. We have observed a similar phenomenon with worst-case top trees, but less pronounced: after all, the top tree interface requires the tree to be partially rebuilt even when only *exposes* are performed.

It should be noted that in all experiments the authors carefully choose the inputs so as to guarantee that no vertex would ever have degree greater than a constant (four or eight, depending on the experiment) in the tree. This is necessary because RC-trees only support vertices of bounded degree. Top trees have no such constraint.

5.8 Final Remarks

With the experiments in this chapter, our goal was to provide a general overview of the strengths and weaknesses of the data structures presented in Chapters 3 and 4. Although the results might change slightly if different implementations are tested or if a different architecture is used, some general observations can be made.

The first is that, for path operations on graphs with small diameter, none of the existing $O(\log n)$ dynamic tree data structures is competitive with the trivial linear-time solution. Even for graphs with $\Theta(\sqrt{n})$ diameter and tens of thousands of vertices, LIN-V and LIN-E are the fastest data structures. On random graphs, the linear-time algorithm was always the fastest.

Even though top trees are much more general (and easy to use) than either ST-trees or ET-trees, their running times are not much worse. Self-adjusting top trees, in particular, were usually no more than a factor of 3 slower than these more specific data structures (except on graphs with very high degree); they were occasionally faster. The worst-case version of top trees was up to eight times slower than ST-trees, particularly when structural operations were numerous. On the other hand, when the number of queries (*exposes*) is large in comparison to *links* and *cuts*, worst-case top trees are remarkably efficient, being even faster than self-adjusting top trees.

The efficiency of worst-case top trees when just queries are performed suggests an extension to the top tree interface to include an operation $C \leftarrow rootcluster(v)$, which returns the root cluster of the top tree containing v. In the original top tree interface, one would call $expose(v)$ for this purpose, but *expose* also guarantees that v will be an endpoint of the root cluster. In some of the applications we have seen this requirement is unnecessary. For example, when the maximum flow algorithm advances (i.e., an arc (v, w) is added to the forest), we must know the root vertex of the component containing w, since it will be the next vertex to be processed. We implement it by calling $expose(w)$ and looking at the *root* field of the cluster returned. This field would still have the correct information even if the returned cluster did not have w as an endpoint, which means we could call $\textit{rootcluster}(w)$ instead. In the worst-case version, this operation would not need to modify the tree at all. The self-adjusting version still requires the tree to be modified, since a generalized splay must pay for the access, but it would be unnecessary to call *hard expose*, which temporarily transforms up to two *compress* nodes into *rake* nodes near the root.

Chapter 6

Final Remarks

We have presented two new data structures for maintaining dynamic trees. They are the first direct implementations of the generic top tree interface, and as such are very flexible. They can naturally represent rooted or unrooted trees, aggregate information over paths or entire trees, and even support ordered adjacency lists.

The first data structure, contraction-based top trees, can perform each dynamic tree operation in $O(\log n)$ time in the worst case, and is a straightforward application of the concept of tree contraction. At all times, it maintains a round-based contraction of the current forest. Whenever there is a *link* or *cut*, it builds a new contraction in the most natural way: by greedily trying to minimize the damage to the original contraction.

Although the update algorithm is conceptually very simple, the proof that it indeed takes $O(\log n)$ time is rather involved, relying on extensive case analysis, which is somewhat unsatisfying. Fortunately, the profusion of cases is not reflected in the actual implementation of the algorithm. Nevertheless, the implementation is still far from trivial, since it must maintain a significant amount of information. Simplifying both the run-time analysis and the implementation of the algorithm are obvious directions for future research.

Self-adjusting top trees, the second data structure we described, are a step towards simplification. Instead of explicitly maintaining a contraction in a bottom-up fashion, they use what at first may seem to be a completely different technique: path decomposition. The original tree is partitioned into edge-disjoint paths, which are represented by binary trees that are then glued together appropriately. But the data structure still supports the top tree interface, which strictly follows the tree contraction paradigm. In fact, self-adjusting top trees demonstrate that path decomposition and tree contraction, rather than being mutually exclusive paradigms for dynamic trees, can be seen as alternative ways of looking at the same data structure. This is useful because path decomposition leads to a simpler and faster update algorithm, but tree contraction is a much more convenient abstraction for the user.

Unfortunately, much of the simplicity of the self-adjusting data structure results from its use of splaying, which only guarantees good amortized performance. For good worst-case performance, splay trees could probably be replaced by globally biased search trees, but these are very complicated data structures. A promising research topic is to devise an alternative that is simple enough to make worst-case data structures based on path decomposition (ST-trees or top trees) practical. This simpler class of binary trees would not have to be as general as globally biased search trees. Globally biased search trees allow nodes to change their weights arbitrarily, and have no restrictions on the range of allowed weights. Within a dynamic tree application, weights are always between 1 and n, where n is the total number of nodes in the original forest. Moreover, node weights do not change arbitrarily: they only change during *splice*, whose specific properties could be exploited.

Our experimental analysis has shown that, although our data structures are not as fast as ST-trees or ET-trees (which are more restricted), they are still competitive. The self-adjusting version, in particular, is usually within a factor of two of ST-trees and ET-trees. Since the top tree interface makes our data structure the easiest to adapt, it should have practical applications. The importance of adaptability cannot be understated. No known dynamic tree data structure is particularly easy to implement. Once a data structure is implemented, one should be able to use it in as many applications as possible.

In fact, one could even consider extending the top tree interface to support even more applications. The current interface was developed to handle cases in which structural operations change only one edge at a time, as is the case of *link* and *cut*. Some applications do not fall into this category. For example, Langerman's algorithm for the shooter location problem [41] and Kaplan et al.'s algorithm for intersecting intervals with priorities [36], both mentioned in Section 2.1.5, require an operation in which a subset of the children of a vertex acquire a new parent. Both data structures we proposed can easily support this operation, but not with the top tree interface.

Another problem that allows more than one structural change to the tree at a time is the mergeable trees problem [26]. Recall from Section 2.1.6 that this is the problem of maintaining rooted dynamic trees with one additional operation: merging paths so as to preserve heap order among the nodes. Georgiadis et al. have shown that standard dynamic-tree data structures (ST-trees or top trees) can perform each operation, including *merges*, in $O(\log^2 n)$ amortized time. They have also presented a new data structure that supports *links* and *merges* in $O(\log n)$ time, as long as *cuts* are not allowed. Supporting *links*, *cuts*, and *merges* in logarithmic time is an interesting open problem.

Yet another direction for future research is to generalize the data structures presented here in order to deal with inputs that are less restricted than trees. Of course, the ultimate goal would be to deal with arbitrary graphs. A less ambitious goal would be to deal with with something in between these two extremes, such as graphs with bounded treewidth, planar graphs, or even grids.

The case of grids is more easily seen as a *dynamic matrix* problem. Concretely, suppose we are given a matrix M and that we are allowed to perform the following operations: (1) change a single entry of M; (2) add a constant k to all entries in a given contiguous submatrix of M; and (3) find the minimum entry in the matrix. One would like to perform these operations in logarithmic time. Even the special case in which the submatrices in the second operation are always entire rows or entire columns would be useful. For instance, it would speed up the local search algorithm for the k-median and facility location problems presented by Resende and Werneck in [47].

A related problem is that of maintaining the minimum value of an array while supporting an operation that adds a constant value to a subset of its entries. If only contiguous subsets are allowed, a simple binary tree can solve this problem. A more general version is to allow these subsets to be arithmetic progressions: one could specify its first entry, the number of entries, and the common difference. This problem also generalizes the special case of the dynamic matrix problem required by Resende and Werneck.

Another interesting generalization of dynamic trees is the problem of performing path-related operations on a planar graph. Klein [38] has recently presented an efficient solution a special case of this problem: given a vertex t on the boundary of the infinite face of an n-vertex planar graph and an arbitrary vertex s, his data structure can find the distance between s and t in $O(\log n)$ time. The data structure can be built in $O(n \log n)$ time and requires $O(n \log n)$ space. It would be desirable to extend this result to all pairs on a planar graph.

In practice, the logarithmic bound seems close to being achieved for some important classes of graphs. Sanders and Schultes [48] and Goldberg et al. [28] have recently proposed algorithms capable of finding the length of the shortest path between any two points of a real-world road network extremely fast, and they present experimental evidence that the algorithm depends logarithmically on the graph size. Proving that this is indeed the case, whether for any of these algorithms or for a third one, is a challenging open problem.

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[2] U. Acar, 2005. Personal Communication.

[3] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 524–533. SIAM, 2004.

[4] U. A. Acar, G. E. Blelloch, and J. L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 41–54, 2005.

[5] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, algorithms, and applications.* Prentice-Hall, 1993.

[6] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[7] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer-Verlag, 1997.

[8] S. Alstrup, J. Holm, and M. Thorup. On the power and speed of top trees. Unpublished manuscript, 1999.

[9] S. Alstrup, J. Holm, and M. thorup. Maintaining center and median in dynamic trees. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1851 of *Lecture Notes in Computer Science*, pages 46–56. Springer-Verlag, 2000.

[10] S. Alstrup, J. Holm, M. Thorup, and K. de Lichtenberg. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

[11] R. Anderson. The `washington` graph generator. In D. S. Johnson and C. C. McGeoch, editors, *DIMACS Series in Discrete Mathematics and Computer Science*, pages 580–581. AMS, 1993. Available at *http://www.avglab.com/andrew/CATS/gens/washington/*.

[12] R. Bellman. On a routing problem. *Quarterly Mathematics*, 16:87–90, 1958.

[13] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14(3):545–568, 1985.

[14] B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2(3):337–361, 1987.

[15] B. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest path algorithms: Theory and experimental evaluation. *Mathematical Programming Series A*, 73(2):129–174, 1996.

[16] R. F. Cohen and R. Tamassia. Dynamic expression trees. *Algorithmica*, 13(3):245–265, 1995.

[17] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[18] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[19] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[20] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[21] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264.

[22] J. Feigenbaum and R. E. Tarjan. Two new kinds of biased search trees. *The Bell System Technical Journal*, 62(10):3139–3158, 1983.

[23] G. N. Frederickson. Data structures for on-line update of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14(4):781–798, 1985.

[24] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal of Computing*, 26(2):484–538, 1997.

[25] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24(1):37–65, 1997.

[26] L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Design of data structures for mergeable trees. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 394–403, 2006.

[27] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Mathematical Programming*, 50:277–290, 1991.

[28] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 129–143. SIAM, 2006.

[29] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[30] D. Goldfarb and J. Hao. A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time. *Mathematical Programming*, 47:353–365, 1990.

[31] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

[32] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarihmic time per operation. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 519–527, 1997.

[33] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarihmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[34] J. Holm and K. de Lichtenberg. Top-trees and dynamic graph algorithms. Technical Report DIKU-TR-98/17, Department of Computer Science, University of Copenhagen, 1998.

[35] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[36] H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic rectangular intersection with priorities. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 639–648, 2003.

[37] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[38] P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 146–155, 2005.

[39] J. Kómlos. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[40] J. B. Kruskal. On the shortest spanning tree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[41] S. Langerman. On the shooter location problem: Maintaining dynamic circular-arc graphs. In *Proceedings of the 12th Canadian Conference on Computational Geometry (CCCG)*, pages 29–35, 2000.

[42] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[43] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 478–489, 1985.

[44] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, 2(1):33–43, 1973.

[45] M. Pătraşcu and E. D. Demaine. Lower bounds for dynamic connectivity. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 546–553, 2004.

[46] T. Radzik. Implementation of dynamic trees with in-subtree operations. *ACM Journal of Experimental Algorithmics*, 3(9), 1998.

[47] M. G. C. Resende and R. F. Werneck. On the implementation of a swap-based local search procedure for the p-median problem. In R. E. Ladner, editor, *Proceedings of the 5th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 119–127. SIAM, 2003.

[48] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th European Symposium on Algorithms (ESA)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[49] R. Sedgewick. *Algorithms in C.* Addison-Wesley, third edition, 1998.

[50] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[51] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[52] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[53] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[54] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM Press, Philadelphia, PA, 1983.

[55] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[56] R. E. Tarjan. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:169–177, 1997.

[57] R. E. Tarjan and R. F. Werneck. Self-adjusting top trees. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 813–822, 2005.

Index

active cluster, 47

admissible network, 119

amortized vs. actual cost, 101

anchor, 50

articulation node, 101

augmented star, 138

augmented top tree, 87

augmenting path, 118

biased search trees, 2

    globally vs. locally, 9

blocking flow, 117

call-back functions

    in RC-trees, 23

    in top trees, 29

center (of a tree), 31

circular order, 24

    always counterclockwise, 24

    eliminating, 80

cluster

    expansion, 69

    in RC-trees, 21

    in top trees, 25

    in topology trees, 18

compress

    in RC-trees, 22

    in topology trees, 19

    parallel, 16

compress tree, 86

contraction, 2

    original vs. new, 47

    parallel, 16

core, 49

    expansion, 69

    size, 67

core image, 49

    size, 67

coupled subtours, 56

create, 29

dashed edges

    in ST-trees, 7

    in top trees, 106

destroy, 29

difference form

    in ET-trees, 35

    in ST-trees, 10

    in topology trees, 20

reverse bit, 11

vs. lazy values, 32

distance label

for maximum flows, 119

for shortest paths, 144

dynamic connectivity, 5

dynamic minimum spanning tree, 21

ET-trees, 2

ET-S, 115

Euler tour

in ET-trees, 34

in top trees, 49, 71

evert, 11

expose

by creating a temporary top tree, 78

by rebuilding the top tree, 77

hard, 97

in ST-trees, 7

soft vs. hard, 91

exposed vertex (in ST-trees), 8

expression trees, 17

dynamic, 20

foster child, 86

guarded splay, 93

handle

in contraction-based top trees, 76

in self-adjusting top trees, 90

hard expose, 97

inactive cluster, 47

internal vertex, 28

join, 29

lazy values, 32

LIN-E, 114

link-cut trees, *see* ST-trees

LIN-V, 114

maximum flow, 20, 32, 118

maximum-priority interval, 15, 158

median (of a tree), 31

mergeable trees, 15, 158

middle child, 2, 7

multilevel partition, 18

network simplex, 14

non-local search, 31

online minimum spanning forest, 128

outer neighbors, 86

outer path, 94

parent set (of a subtour), 52

phantom tree, 14

of a top tree, 101

potential (of a phantom tree), 101

proper child, 86

rake

in RC-trees, 22

in top trees, 25

in topology trees, 19, 25

parallel, 16

rake tree, 86, 138

random layered graph, 124

rank (of a top tree node), 101

RC-trees, 3, 21, 152

alternating rounds, 83

compress, 22

rake, 22

randomization, 83

rectification, 93, 95, 100

residual capacity, 118

residual network, 118

reverse bit

in ST-trees, 11

in top trees, 90

root path, 85

rotation

in ST-trees, 8

within compress trees, 93

saturated arc, 118

select, 31

self-adjusting top trees, 85

shadow tree, 7

shooter location problem, 14, 158

soft expose, 91

solid paths (in ST-trees), 6

solid subtree, 7

splay trees, 2, 9, 35

splaying

guarded, 92

within self-adjusting top trees, 92

splaying split, 96

splice

in ST-trees, 8

in top trees, 94

split, 29

ST-trees, 2, 6

ST-E, 114

ST-V, 113

subtour

inactive, 50

proper, 50

stable, 55

unstable, 56

ternarization, 2, 3, 13

top trees, 3, 151

self-adjusting, 85

topology trees, 3, 18, 151

compress, 19

rake, 19

TOP-S, 115

TOP-W, 115

tree contraction, *see* contraction

twin arcs, 49

unit tree, 85

virtual tree, 7